

SESSION 7

---

# iOS DEVELOPMENT

## ADVANCED IOS

- ▶ It's time to start thinking about courses for next quarter!
- ▶ Advanced iOS (MPCS 51032) will be offered on Tuesday evenings in the Spring

# ADVANCED IOS

- ▶ Topics that have been covered in the past
  - ▶ Integrating with Siri
  - ▶ Building apps for Apple Watch
  - ▶ Building games with SpriteKit
  - ▶ Building AR experiences

---

# CASE STUDIES

## CASE STUDY PRESENTATIONS

- ▶ Case studies will be on Wednesday, March 4
- ▶ 5 minute presentation on an app in the App Store
  - ▶ Plus a few extra minutes for Q&A
- ▶ You may work individually or with one partner

## CASE STUDY PRESENTATIONS

- ▶ Add your name and the app you're presenting on to the sign-up sheet
- ▶ Don't sign up for an app someone else is presenting on!

# CASE STUDY PRESENTATIONS

- ▶ Basic information:
  - ▶ What is the basic functionality and purpose of the app?
  - ▶ Who is the developer? What role does the app play in their business model?
  - ▶ How long has the app been around? How does it rank on the App Store?

## CASE STUDY PRESENTATIONS

- ▶ Competition:
  - ▶ Who are the app's primary competitors?
  - ▶ How does it set itself apart from the competition?

# CASE STUDY PRESENTATIONS

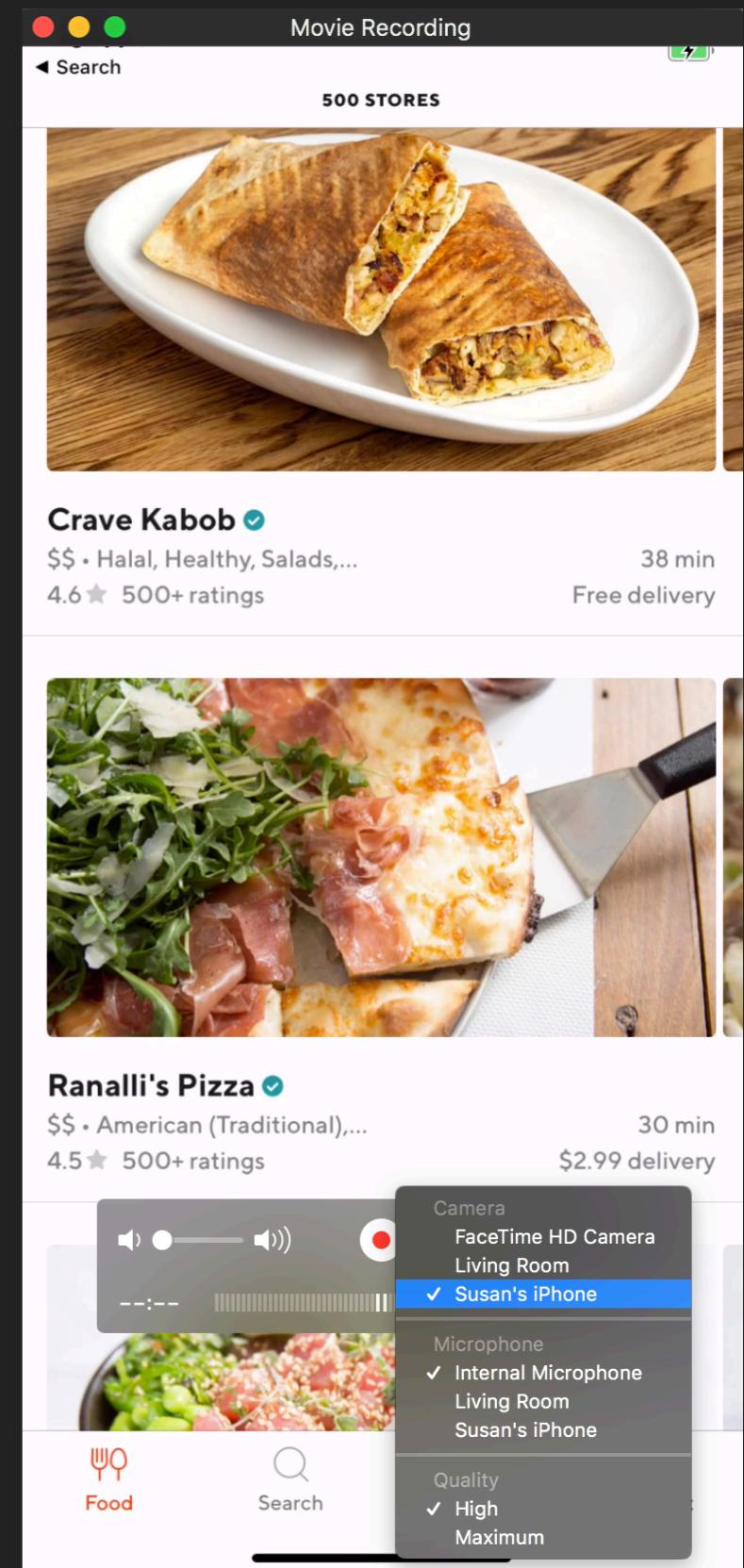
- ▶ Functionality:
  - ▶ Describe the app's central functionality
  - ▶ Show screenshots and / or videos
  - ▶ What do you like about this app? What could be improved?

# CASE STUDY PRESENTATIONS

- ▶ Technical Implementation:
  - ▶ Tell us which UI components and frameworks you think were used
  - ▶ Choose a feature that looks challenging to implement and describe how it might have been done

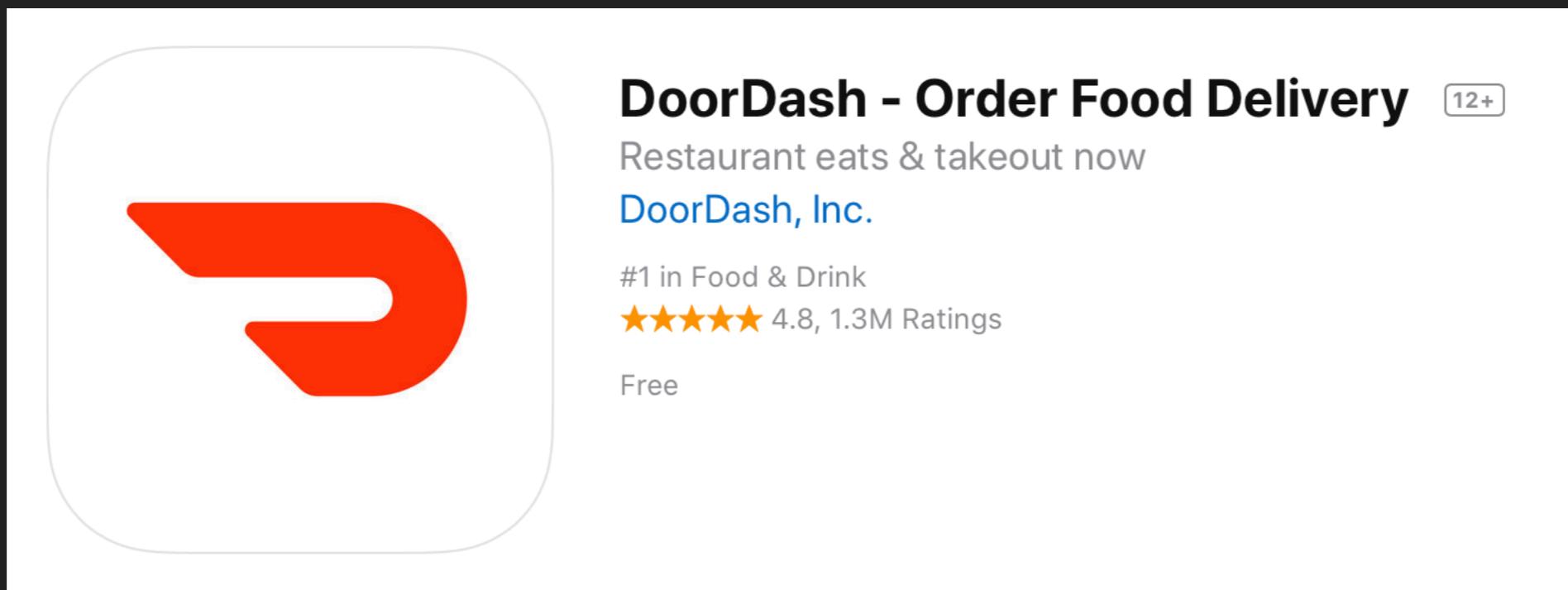
# CREATING VIDEOS

- ▶ To record a video of an app:
  - ▶ Connect your iOS device to your laptop
  - ▶ Open Quicktime and select "New Movie Recording"
  - ▶ Choose your device from the dropdown next to the record button



# DOORDASH

- ▶ Food delivery service
- ▶ Founded in 2013 by four students at Stanford University
- ▶ Ranked #1 in Food & Drink category on the App Store



# CASE STUDIES

## DOORDASH

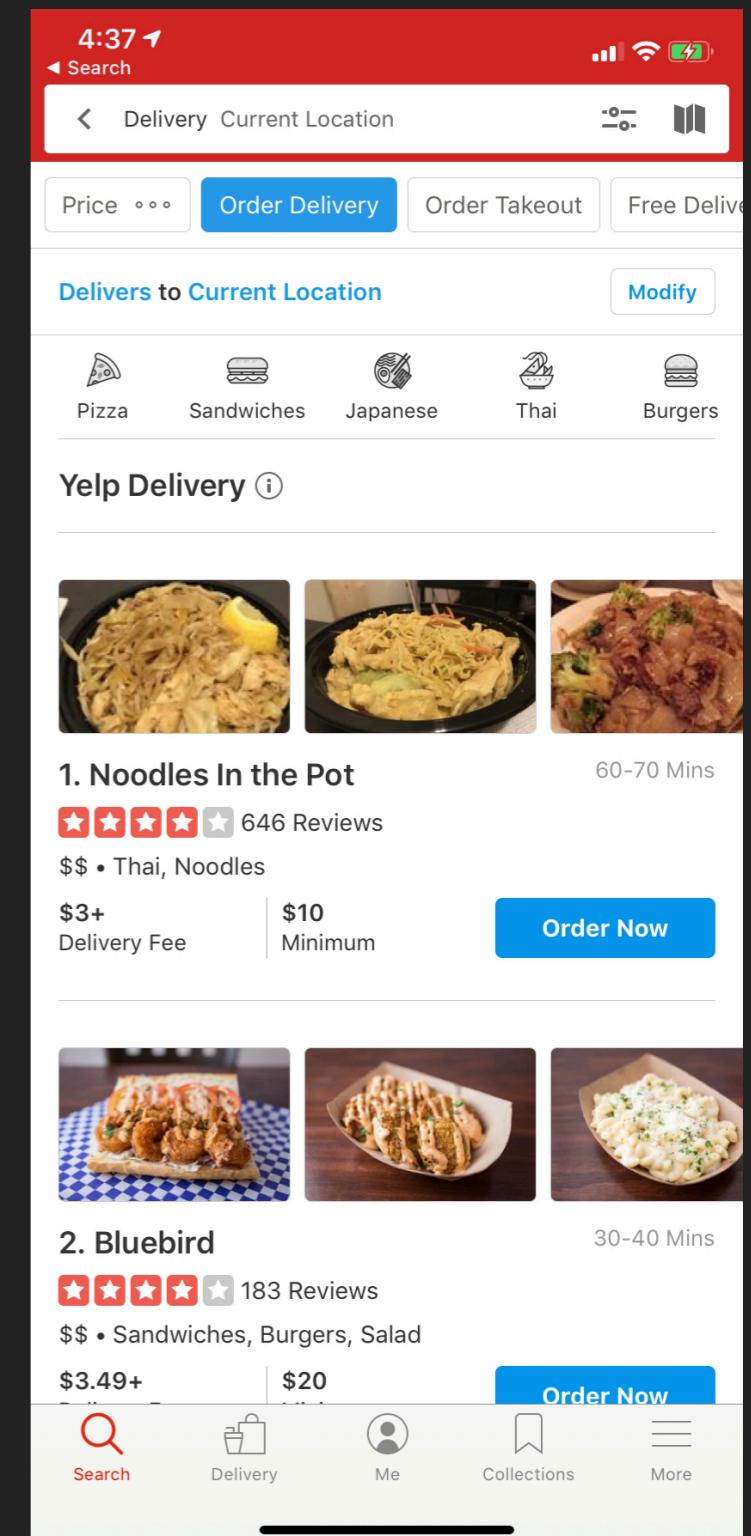
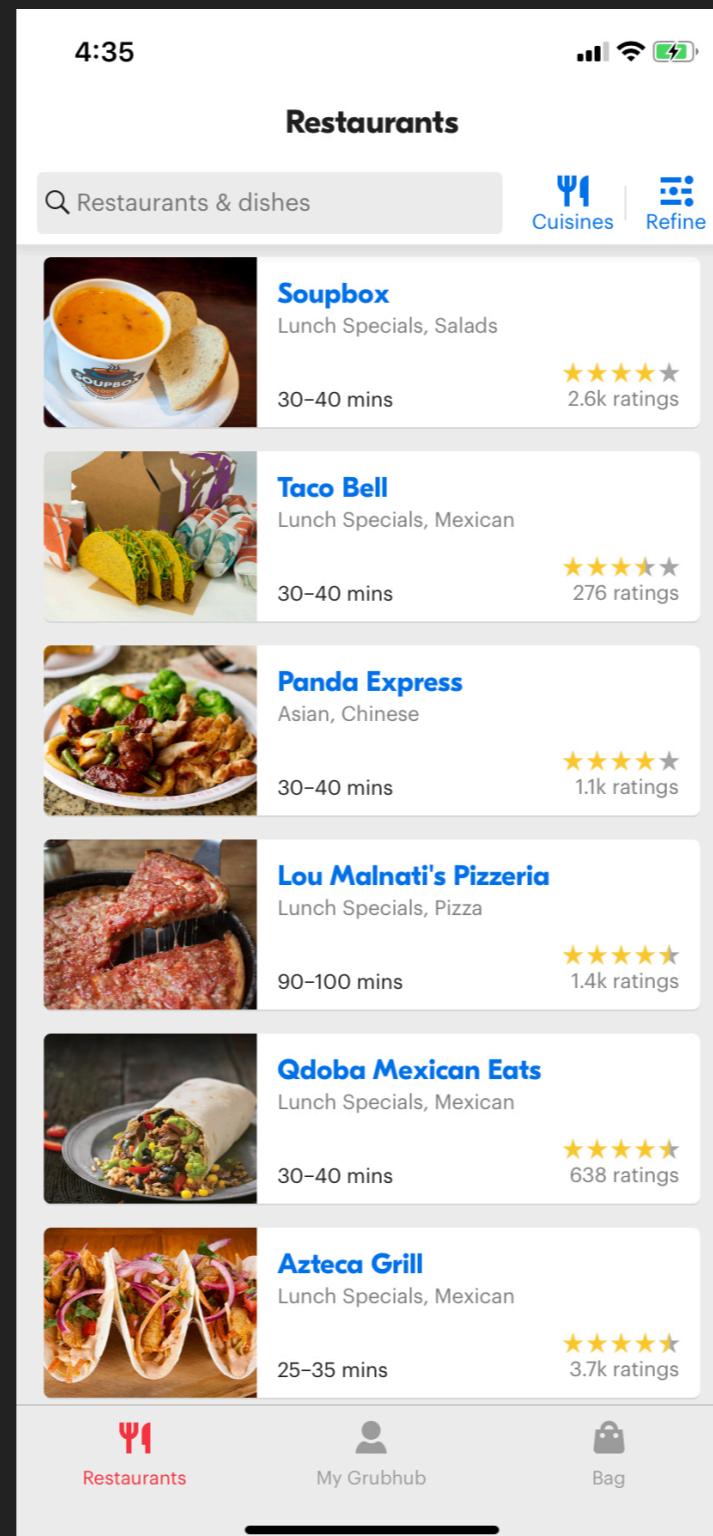
► Competition:

► Grubhub

► Yelp

► Postmates

► Uber Eats



Grubhub

Yelp

## CASE STUDIES

## DOORDASH

- ▶ Uses Apple's preferred design patterns, like large titles
- ▶ More intuitive and visually pleasing than it's competitors

[Back](#)

## Nando's PERi-PERi ✓

\$ • Portuguese, African, Chicken

★★★★★ newly added

**\$3.99 28 0.8**

delivery min miles

## FEATURED ITEMS

**1/2 Chicken (2 Regular Sides)**

\$15.55

[FULL MENU](#)[Switch Menu](#)

Popular items

10 &gt;

Fire Starters

5 &gt;

PERi-PERi Chicken

18 &gt;



Food



Drinks



Search



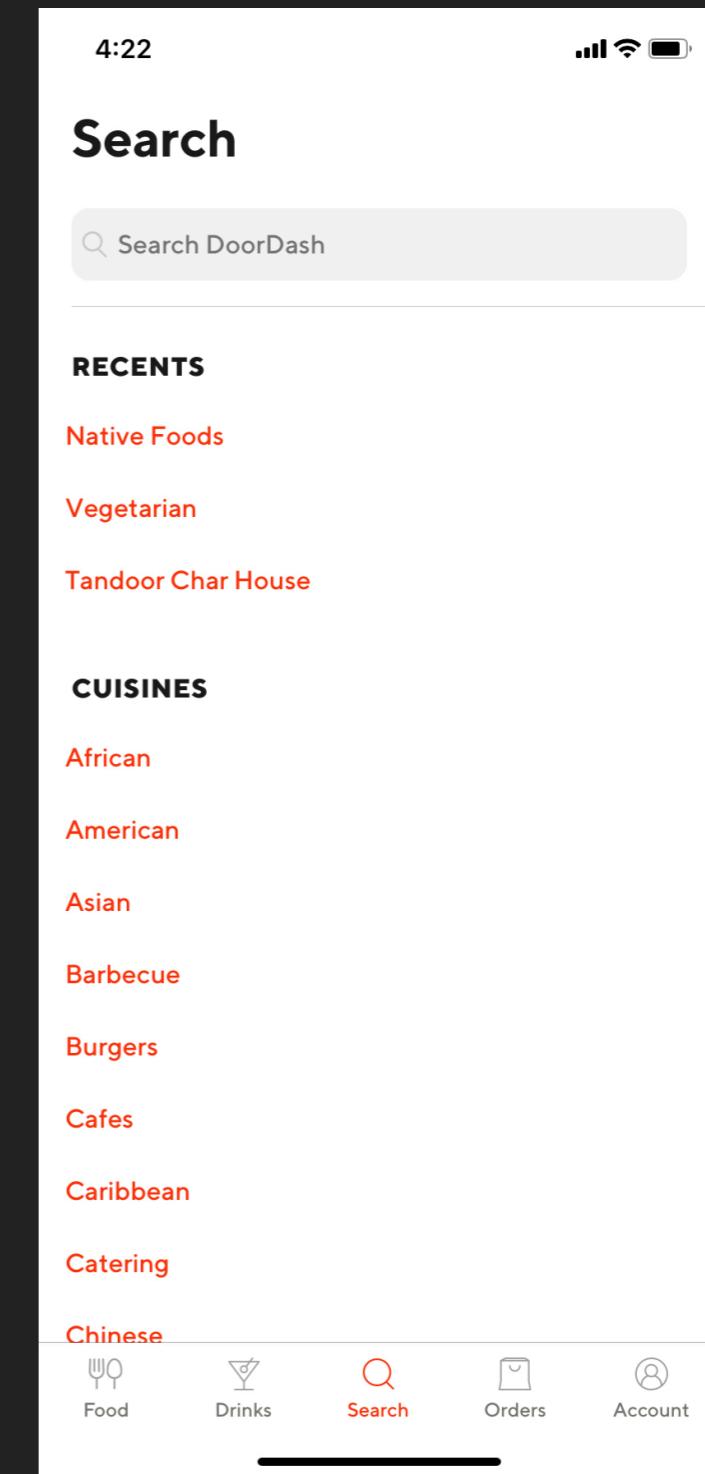
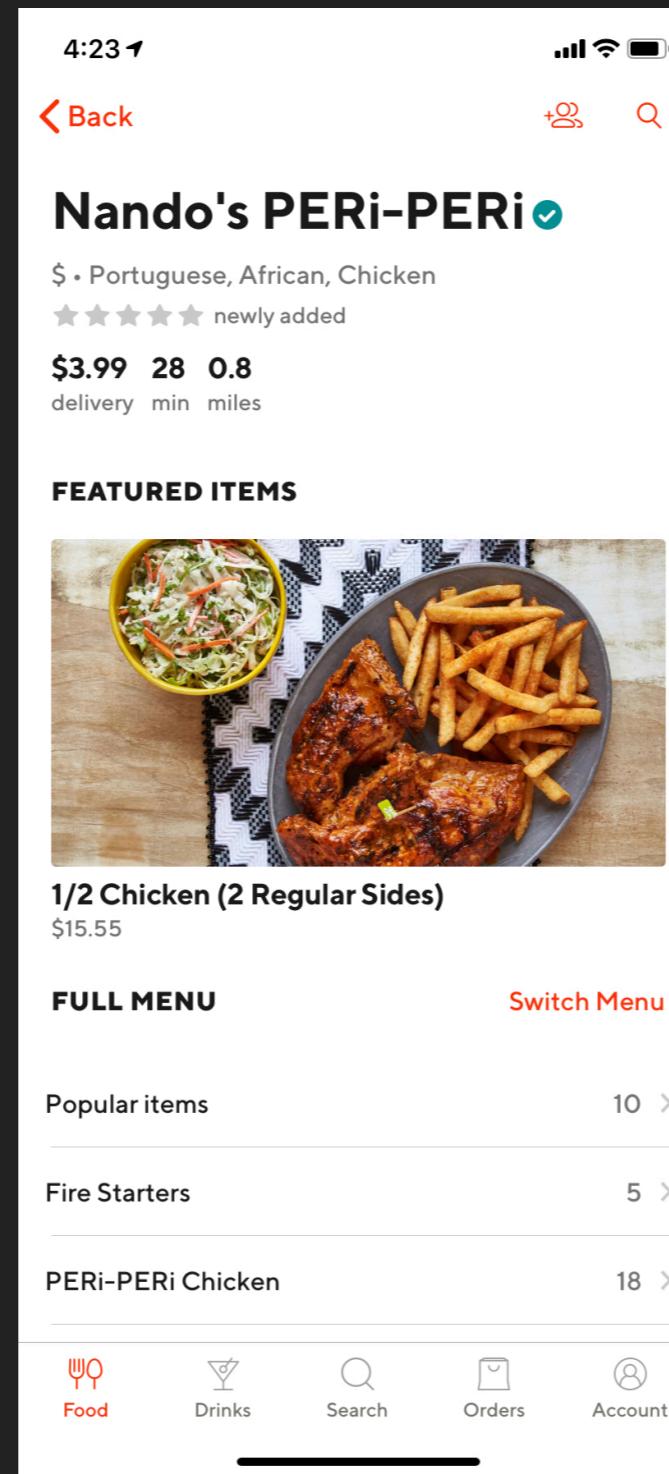
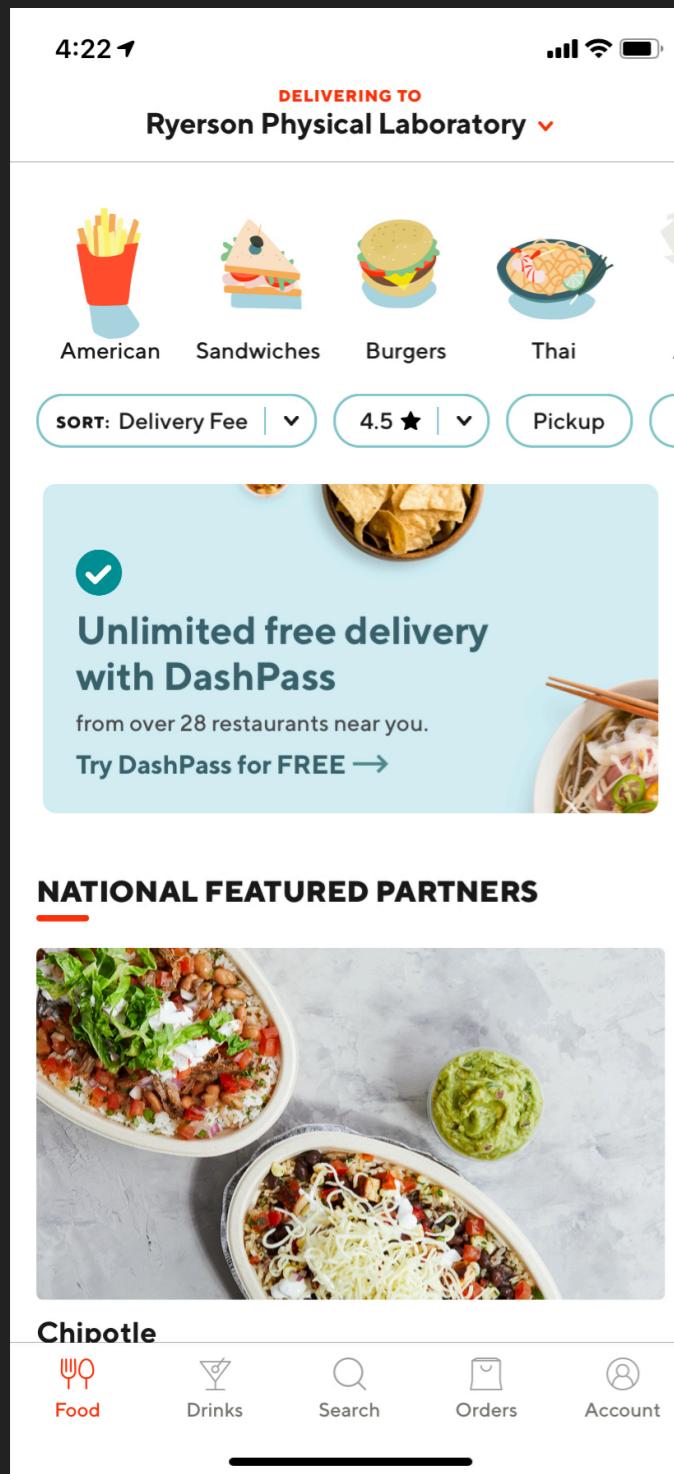
Orders



Account

# CASE STUDIES

## DOORDASH



## CASE STUDIES

### DOORDASH

- ▶ Full-screen view controller presented modally
- ▶ Dark gray view with alpha < 1.0
- ▶ Animates on-screen in `viewDidAppear`

9:41

DELIVERING TO  
Ryerson Physical Laboratory ▾



American



Sandwiches



Burgers

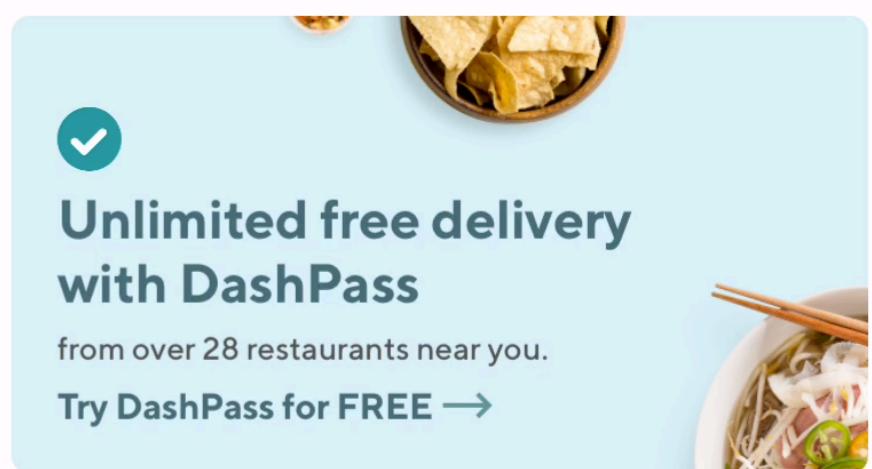


Thai

SORT: Delivery Fee ▾

4.5 ★ ▾

Pickup



#### NATIONAL FEATURED PARTNERS



Chipotle

Food

Drinks

Search

Orders

Account

AN INTRODUCTION TO

---

# SWIFTUI

# INTRODUCING SWIFTUI

- ▶ Apple introduced SwiftUI at WWDC 2019
  - ▶ Swift-only UI framework
  - ▶ Alternative to UIKit

# COMPARISON

- ▶ UIKit
  - ▶ Obj C and Swift
  - ▶ Storyboards
  - ▶ Any version of iOS
  - ▶ Imperative
- ▶ SwiftUI
  - ▶ Swift only
  - ▶ Live Previews
  - ▶ iOS 13 and above
  - ▶ Declarative

## DECLARATIVE VS. IMPERATIVE

- ▶ With **imperative** programming, developers give step-by-step instructions
  - ▶ “First make a label. Then add it as a subview...”
- ▶ With **declarative** programming, developers describe the desired end-state
  - ▶ “I want a label in the center of the screen.”

## IMPERATIVE EXAMPLE

- ▶ Imagine teaching someone who doesn't know how to cook to make pasta over the phone:
  - ▶ 1. Boil water on the stove.
  - ▶ 2. Add the pasta to the water.
  - ▶ 3. Set a timer for 10 minutes.
  - ▶ 4. Check if the pasta is done.
  - ▶ 5. Etc...

## DECLARATIVE EXAMPLE

- ▶ Imagine ordering pasta at a restaurant:
  - ▶ “I’d like spaghetti with meatballs and extra Parmesan cheese.”

# VIEWS

- ▶ Views are the building blocks of user interfaces
  - ▶ In UIKit, we use **UIView**
  - ▶ In SwiftUI, we use **View**

# Views

- ▶ Here is the code for a basic view in SwiftUI

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}
```

# VIEWS

- ▶ Here is the code for a basic view in SwiftUI

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}
```

# Views

- Here is the code for a basic view in SwiftUI

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}
```

View is a protocol,  
not a class

# VIEWS

- ▶ Here is the code for a basic view in SwiftUI

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}
```

Every view has a  
**body**, which is  
itself a view

# VIEWS

- Here is the code for a basic view in SwiftUI

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}
```

Notice that we  
don't need to call  
`addSubview`

# VIEW MODIFIERS

- ▶ Change how a view renders using **view modifiers**

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
            .font(Font.largeTitle)
    }
}
```

Font modifier

# VIEW MODIFIERS

- ▶ The **font** modifier is a method on **Text**
- ▶ It takes font parameter and returns an instance of **Text**

The screenshot shows a web browser window with the Apple Developer website open. The URL bar shows the path: Documentation > ... > Text > font(\_:). The main content area is titled "Instance Method" and features a large bold heading "font(\_:)". A descriptive text below it states "Sets the default font for text in the view." A horizontal line separates this from the "Declaration" section, which contains the Swift code "func font(\_ font: Font?) -> Text". Another horizontal line separates the declaration from the "Parameters" section, which includes a "font" parameter description. A final horizontal line separates the parameters from the "Return Value" section, which describes the output as "Text that uses the font you specify".

Documentation > ... > Text > font(\_:)

Instance Method

## font(\_:)

Sets the default font for text in the view.

---

## Declaration

```
func font(_ font: Font?) -> Text
```

---

## Parameters

**font**  
The font to use when displaying this text.

---

## Return Value

Text that uses the font you specify.

# VIEW MODIFIERS

- ▶ You can chain modifiers together, like this:

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
            .font(Font.largeTitle)
            .foregroundColor(Color.purple)
    }
}
```

Chaining view  
modifiers

# VIEW MODIFIERS

- ▶ The `PreviewProvider` protocol allows you to preview views as you design them

The screenshot shows an Xcode interface with a dark theme. The top menu bar includes File, Edit, View, Find, Navigate, Editor, Product, Debug, Source Control, Window, and Help. The status bar at the bottom right shows the date as Feb 17 5:03 PM and battery level as 5%.

The main area displays `ContentView.swift` code:

```
8
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         Text("Hello, World!")
14             .font(.largeTitle)
15             .foregroundColor(Color.purple)
16     }
17 }
18
19 struct ContentView_Previews: PreviewProvider {
20     static var previews: some View {
21         ContentView()
22     }
23 }
```

A red rectangular box highlights the `ContentView_Previews` struct and its implementation. A red arrow points from this highlighted code to the preview window on the right.

The preview window shows a black iPhone 11 Pro Max simulator with a white background. Inside the screen, the text "Hello, World!" is displayed in a large, bold, purple font.

# VIEW MODIFIERS

- ▶ The preview updates automatically as you change your code

The screenshot shows the Xcode interface with the following details:

- File Menu:** Xcode, File, Edit, View, Find, Navigate, Editor, Product, Debug, Source Control, Window, Help.
- Toolbar:** Standard Xcode toolbar with icons for file operations.
- Project Navigator:** Shows the project structure: SwiftUIDemo > SwiftUIDemo > ContentView.swift.
- Editor:** Displays the `ContentView.swift` file content. A red box highlights the line `.background(Color.yellow)`. The code is as follows:

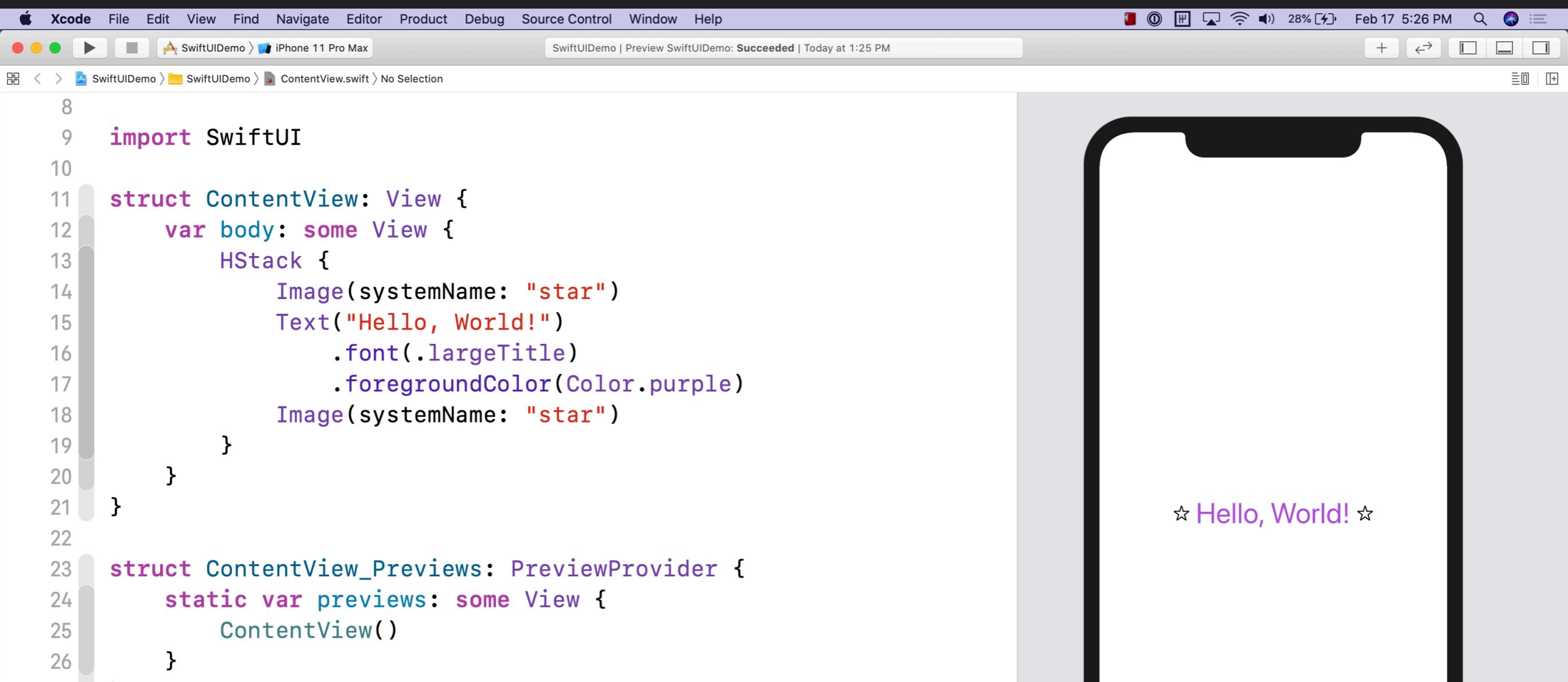
```
8
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         Text("Hello, World!")
14             .font(.largeTitle)
15             .foregroundColor(Color.purple)
16             .background(Color.yellow)
17     }
18 }
19
20 struct ContentView_Previews: PreviewProvider {
21     static var previews: some View {
22         ContentView()
23     }
24 }
```
- Preview Area:** Shows a preview of the iPhone 11 Pro Max screen displaying the text "Hello, World!" in purple font on a yellow background.
- StatusBar:** Shows the date and time as Feb 17 5:06 PM.

## VIEW MODIFIERS

- ▶ SwiftUI provides three container views that arrange other views into stacks
  - ▶ HStack (horizontal)
  - ▶ VStack (vertical)
  - ▶ ZStack (overlay)

# VIEW MODIFIERS

- ▶ HStack arranges its child views horizontally



The screenshot shows the Xcode interface with the following details:

- Top Bar:** Xcode, File, Edit, View, Find, Navigate, Editor, Product, Debug, Source Control, Window, Help.
- Toolbar:** Shows icons for running, stopping, and previewing.
- Project Navigator:** Shows the project structure: SwiftUIDemo > SwiftUIDemo > ContentView.swift.
- Editor:** Displays the `ContentView.swift` file content. The code uses `HStack` to arrange two `Image` views and one `Text` view horizontally.
- Preview:** A simulator window shows the resulting UI on an iPhone 11 Pro Max, featuring two purple star icons flanking a large red "Hello, World!" text.
- Status Bar:** Shows the date (Feb 17), time (5:26 PM), battery level (28%), and signal strength.

```
8
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         HStack {
14             Image(systemName: "star")
15             Text("Hello, World!")
16                 .font(.largeTitle)
17                 .foregroundColor(Color.purple)
18             Image(systemName: "star")
19         }
20     }
21 }
22
23 struct ContentView_Previews: PreviewProvider {
24     static var previews: some View {
25         ContentView()
26     }
}
```

# VIEW MODIFIERS

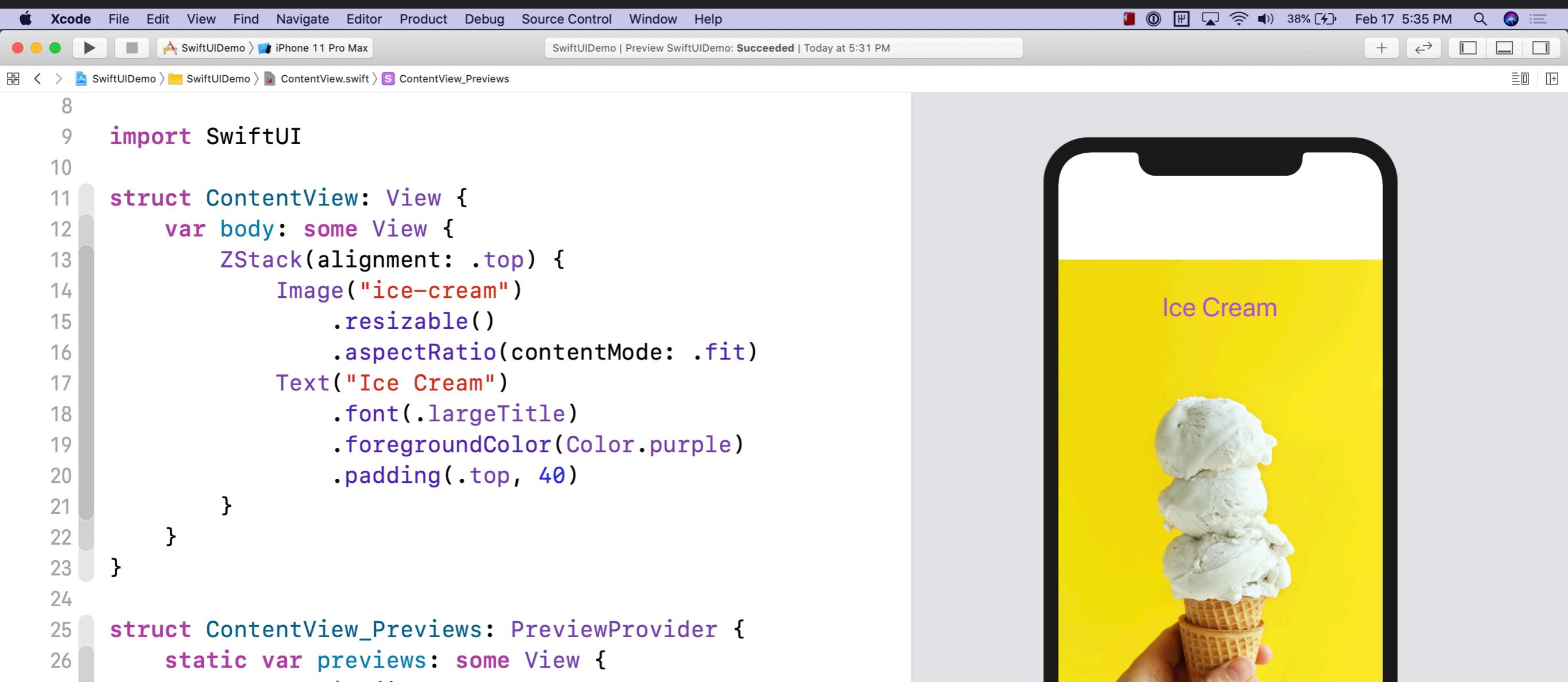
- ▶ VStack arranges its child views vertically

The screenshot shows the Xcode interface with the ContentView.swift file open. The code defines a struct ContentView with a body containing a VStack view. The VStack has an alignment modifier set to .leading. Inside the VStack, there are two Text views: one with the text "iOS Development" and another with "MPCS 51030". The "iOS Development" text is styled with a large title font and purple color. A red callout box with the text "Leading alignment" and a curved arrow points to the alignment modifier in the code. To the right of the code editor is a simulator window showing an iPhone 11 Pro Max displaying the same text content.

```
8  
9 import SwiftUI  
10  
11 struct ContentView: View {  
12     var body: some View {  
13         VStack(alignment: .leading) {  
14             Text("iOS Development")  
15                 .font(.largeTitle)  
16                 .foregroundColor(Color.purple)  
17             Text("MPCS 51030")  
18         }  
19     }  
20 }  
21  
22 struct ContentView_Previews: PreviewProvider {  
23     static var previews: some View {  
24         ContentView()  
25     }  
26 }
```

# VIEW MODIFIERS

- ▶ ZStack overlays its child views on top of each other



The screenshot shows the Xcode interface with the following details:

- File Navigator:** Shows the project structure: SwiftUIDemo > SwiftUIDemo > ContentView.swift.
- Editor:** Displays the code for `ContentView` and `ContentView_Previews`.
- Preview:** A preview window shows an iPhone 11 Pro Max displaying a yellow background with a stack of white ice cream scoops in a waffle cone. The text "Ice Cream" is displayed in purple at the top right of the screen.
- Top Bar:** Includes the Xcode logo, file menu, and various status icons like battery level (38%) and signal strength.
- Bottom Bar:** Shows the date and time (Feb 17 5:35 PM) and system controls.

```
8
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         ZStack(alignment: .top) {
14             Image("ice-cream")
15                 .resizable()
16                 .aspectRatio(contentMode: .fit)
17             Text("Ice Cream")
18                 .font(.largeTitle)
19                 .foregroundColor(Color.purple)
20                 .padding(.top, 40)
21         }
22     }
23 }
24
25 struct ContentView_Previews: PreviewProvider {
26     static var previews: some View {
27         ContentView()
28     }
29 }
```

# VIEW MODIFIERS

- ▶ Notice how the code mimics the view hierarchy

The screenshot shows the Xcode interface with the following details:

- File Navigator:** Shows the project structure: SwiftUIDemo > SwiftUIDemo > ContentView.swift.
- Editor:** Displays the `ContentView.swift` file content:8  
9 import SwiftUI  
10  
11 struct ContentView: View {  
12 var body: some View {  
13 VStack(alignment: .leading) {  
14 Text("iOS Development")  
15 .font(.largeTitle)  
16 .foregroundColor(Color.purple)  
17 Text("MPCS 51030")  
18 }  
19 }  
20 }  
21  
22 struct ContentView\_Previews: PreviewProvider {  
23 static var previews: some View {  
24 ContentView()  
25 }  
26 }
- Preview Area:** Shows the iPhone 11 Pro Max preview with the text "iOS Development" in large purple font and "MPCS 51030" below it.
- Annotations:** A blue box labeled "VSTACK" has arrows pointing to the `VStack` modifier in the code and to the first `Text` element in the preview. Another blue box labeled "TEXT" has arrows pointing to the two `Text` elements in the code and to their corresponding text in the preview.

LISTS AND NAVIGATION

---

SWIFTUI

# LISTS

- ▶ **List** allows you to present multiple rows of data

The screenshot shows the Xcode interface with the ContentView.swift file open. The code defines a struct ContentView that contains a List of five items, each showing an emoji and the text "Face With Starry Eyes". A red callout box with the text "Looks like a table view!" points from the code to the simulator window.

```
8
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         List(0..<5) { item in
14             Image("grinning-face-with-star-eyes")
15                 .resizable()
16                 .frame(width: 60, height: 60)
17             Text("Face With Starry Eyes")
18         }
19     }
20 }
21
22 struct ContentView_Previews: PreviewProvider {
23     static var previews: some View {
24         ContentView()
25     }
26 }
```

Running SwiftUIDemo on iPhone 11 Pro Max

Face With Starry Eyes

# LISTS

- ▶ List allows you to present multiple rows of data

The image shows a screenshot of an Xcode workspace. The main area displays the `ContentView.swift` file:

```
8
9 import SwiftUI
10
11 struct ContentView: View {
12     var body: some View {
13         List(0..<5) { item in
14             Image("grinning-face-with-star-eyes")
15                 .resizable()
16                 .frame(width: 60, height: 60)
17             Text("Face With Starry Eyes")
18         }
19     }
20 }
21
22 struct ContentView_Previews: PreviewProvider {
23     static var previews: some View {
24         ContentView()
25     }
26 }
```

A red box highlights the code within the `List` block, specifically the `Image` and `Text` components. A red callout box with the text "Views displayed in each row" points to this highlighted code. To the right of the Xcode window is an iPhone 11 Pro Max simulator showing the resulting UI. The screen displays five identical rows, each consisting of a yellow emoji with star-shaped eyes followed by the text "Face With Starry Eyes".

Row	Image	Text
1	Face With Starry Eyes	Face With Starry Eyes
2	Face With Starry Eyes	Face With Starry Eyes
3	Face With Starry Eyes	Face With Starry Eyes
4	Face With Starry Eyes	Face With Starry Eyes
5	Face With Starry Eyes	Face With Starry Eyes

# LISTS

- ▶ List allows you to present multiple rows of data

The image shows a screenshot of the Xcode IDE. The top bar displays the menu: Xcode, File, Edit, View, Find, Navigate, Editor, Product, Debug, Source Control, Window, Help. The status bar shows the date and time: Feb 17 7:04 PM. The main area shows the ContentView.swift file:

```
8  
9 import SwiftUI  
10  
11 struct ContentView: View {  
12     var body: some View {  
13         List(0..<5) { item in  
14             Image("grinning-face-with-star-eyes")  
15                 .resizable()  
16                 .frame(width: 60, height: 60)  
17             Text("Face With Starry Eyes")  
18         }  
19     }  
20 }  
21  
22 struct ContentView_Previews: PreviewProvider {  
23     static var previews: some View {  
24         ContentView()  
25     }  
26 }
```

A red box highlights the `List(0..<5) { item in` part of the code. A red arrow points from this box to a red-bordered callout box containing the text "Data for each row". To the right of the code editor is a screenshot of an iPhone 11 Pro Max displaying a list of five items, each consisting of a yellow emoji with star-shaped eyes and the text "Face With Starry Eyes".

# LISTS

- ▶ SwiftUI needs a way to distinguish the model object for each row
  - ▶ Option 1: Provide a **key path** to a property that uniquely identifies each model object
  - ▶ Option 2: Have the model conform to the **Identifiable** protocol

# LISTS

- ▶ Here is our `Emoji` model

```
struct Emoji: Codable {  
    let name: String  
    let year: String  
    let imageName: String  
}
```

# LISTS

- We can pass **name** as the identifier

The screenshot shows the Xcode interface with the ContentView.swift file open. The code defines a struct ContentView that contains a List view. The List view is configured with the emojis array as its source and uses the id: \.name parameter to identify each item. A red box highlights the id: \.name parameter, and a red arrow points from it to a red box labeled "Identifier". The right side of the screen shows an iPhone 11 Pro Max simulator displaying a list of eight emojis, each with its name and a small image.

```
8
9 import SwiftUI
10
11 struct ContentView: View {
12
13     let emojis: [Emoji] = emojiData()
14
15     var body: some View {
16         List(emojis, id: \.name) { emoji in
17             Image(emoji.imageName)
18                 .resizable()
19                 .frame(width: 60, height: 60)
20             Text(emoji.name)
21         }
22     }
23 }
24
```

Emoji	Name
😊	Grinning Face
🤩	Face With Starry Eyes
(zipper mouth)	Face With a Zipper Mouth
😜	Goofy Face
🤠	Cowboy Face
🦊	Fox Face
🐶	Dog Face
🦁	Lion Face

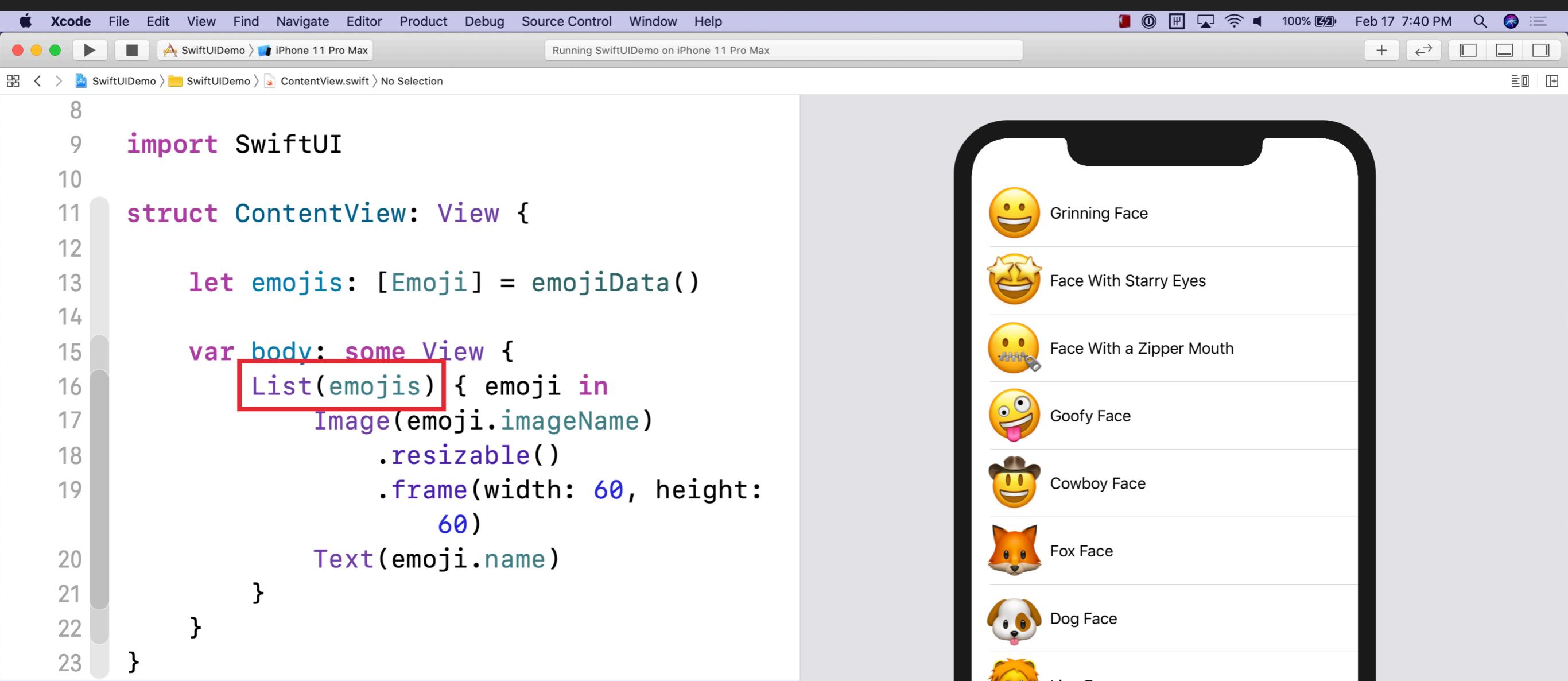
# LISTS

- ▶ Alternatively, we can update `Emoji` to conform to the `Identifiable` protocol

```
struct Emoji: Codable, Identifiable {  
    let name: String  
    let year: String  
    let imageName: String  
    let id: UUID = UUID()  
}
```

# LISTS

- Now we can pass in an array of emojis without needing to provide the `id` key path as well



The screenshot shows the Xcode interface with the ContentView.swift file open. The code defines a struct ContentView that contains a list of emojis. The List view is highlighted with a red box. The right side of the screen shows an iPhone 11 Pro Max simulator displaying a list of emojis with their names.

```
8
9 import SwiftUI
10
11 struct ContentView: View {
12
13     let emojis: [Emoji] = emojiData()
14
15     var body: some View {
16         List(emojis) { emoji in
17             Image(emoji.imageName)
18                 .resizable()
19                 .frame(width: 60, height:
20                         60)
21             Text(emoji.name)
22         }
23     }
}
```

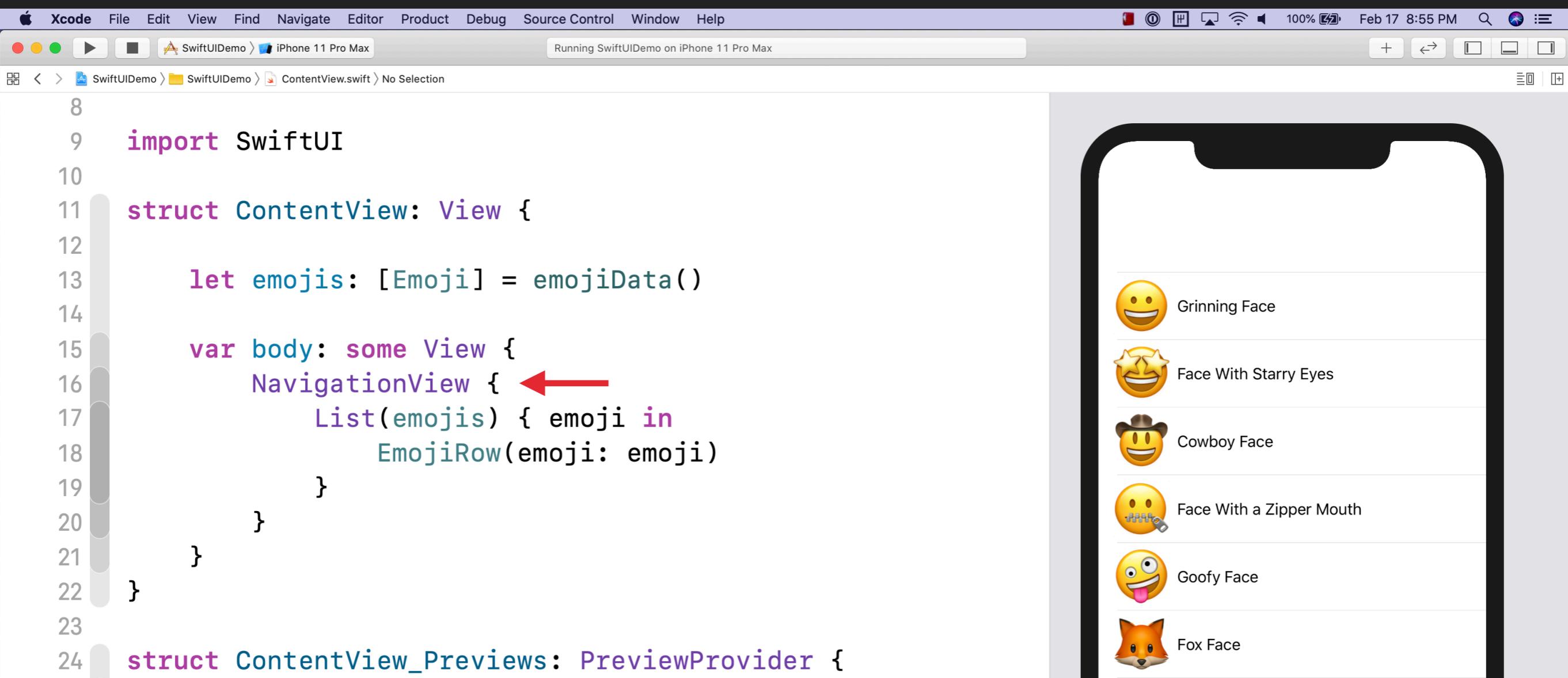
Emoji	Name
😊	Grinning Face
🤩	Face With Starry Eyes
(zip)	Face With a Zipper Mouth
😜	Goofy Face
🤠	Cowboy Face
🦊	Fox Face
🐶	Dog Face
🦁	Lion Face

## NAVIGATION VIEWS

- ▶ Suppose we want users to be able to tap on a row and navigate to a detail page
- ▶ This can be achieved using **NavigationView**

# NAVIGATION VIEWS

- ▶ Embed the list in a `NavigationView`



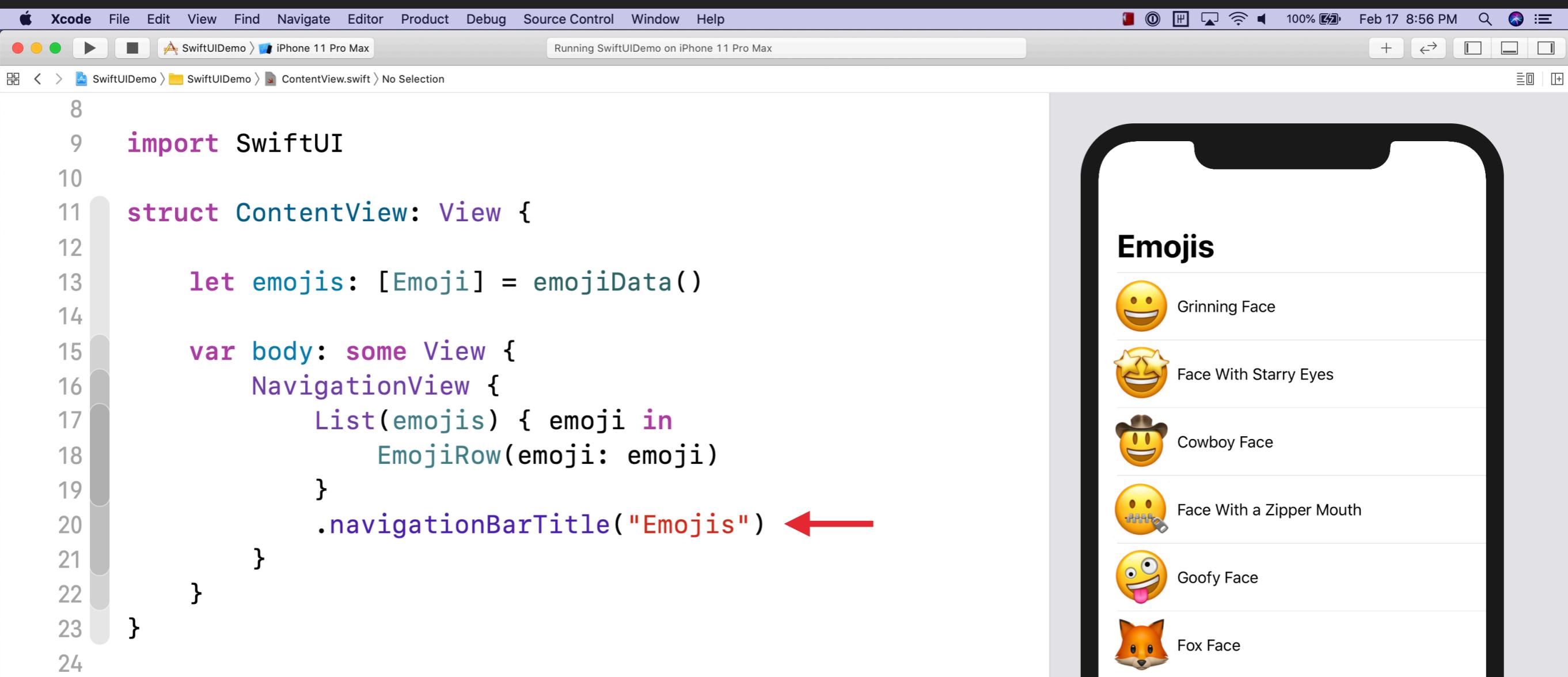
The screenshot shows the Xcode interface with the ContentView.swift file open. A red arrow points to the `NavigationView` opening brace at line 16. To the right is a preview of an iPhone 11 Pro Max displaying a list of emojis.

```
8
9 import SwiftUI
10
11 struct ContentView: View {
12
13     let emojis: [Emoji] = emojiData()
14
15     var body: some View {
16         NavigationView { ←
17             List(emojis) { emoji in
18                 EmojiRow(emoji: emoji)
19             }
20         }
21     }
22 }
23
24 struct ContentView_Previews: PreviewProvider {
```

Emoji	Description
😊	Grinning Face
🤩	Face With Starry Eyes
🤠	Cowboy Face
🤐	Face With a Zipper Mouth
😜	Goofy Face
🦊	Fox Face

# NAVIGATION VIEWS

- ▶ Add a title using the `navigationBarTitle` modifier



The screenshot shows the Xcode interface with the ContentView.swift file open. The code defines a ContentView struct that contains a NavigationView with a List of emojis. A red arrow points to the `.navigationBarTitle("Emojis")` line. To the right, the iPhone 11 Pro Max simulator displays the resulting application. The title "Emojis" is visible at the top of the screen, followed by a list of emoji icons with their names.

```
8
9 import SwiftUI
10
11 struct ContentView: View {
12
13     let emojis: [Emoji] = emojiData()
14
15     var body: some View {
16         NavigationView {
17             List(emojis) { emoji in
18                 EmojiRow(emoji: emoji)
19             }
20             .navigationBarTitle("Emojis") ←
21         }
22     }
23 }
24 }
```

Emoji	Name
😊	Grinning Face
🤩	Face With Starry Eyes
🤠	Cowboy Face
🤐	Face With a Zipper Mouth
😜	Goofy Face
🦊	Fox Face

# NAVIGATION VIEWS

- ▶ Set the `displayMode` to `inline` if you don't want large titles

The screenshot shows the Xcode interface with the ContentView.swift file open. The code defines a struct ContentView that contains a NavigationView with a List of emojis. The navigation bar title is set to "Emojis" with a display mode of .inline. A red underline highlights the `.navigationBarTitle("Emojis", displayMode: .inline)` line. To the right, a preview window shows an iPhone 11 Pro Max displaying a list titled "Emojis" with several emoji cards.

```
8
9 import SwiftUI
10
11 struct ContentView: View {
12
13     let emojis: [Emoji] = emojiData()
14
15     var body: some View {
16         NavigationView {
17             List(emojis) { emoji in
18                 EmojiRow(emoji: emoji)
19             }
20             .navigationBarTitle("Emojis", displayMode: .inline)
21         }
22     }
23 }
24
```

Emoji	Name
😊	Grinning Face
🤩	Face With Starry Eyes
🤠	Cowboy Face
😃	Face With a Zipper Mouth
😜	Goofy Face
🦊	Fox Face
🐶	Dog Face

# NAVIGATION VIEWS

- ▶ Add a `NavigationLink` so that tapping on a row triggers navigation to a new page

The screenshot shows the Xcode interface with the ContentView.swift file open. The code defines a ContentView struct containing a NavigationView with a List of emojis, each wrapped in a NavigationLink that points to a "Hello" text view. The navigation bar title is set to "Emojis". To the right, an iPhone 11 Pro Max simulator displays a list of emojis with their names:

Emoji	Name
😊	Grinning Face
🤩	Face With Starry Eyes
🤠	Cowboy Face
😃	Face With a Zipper Mouth
😜	Goofy Face
🦊	Fox Face
🐶	Dog Face

HANDLING DATA IN

---

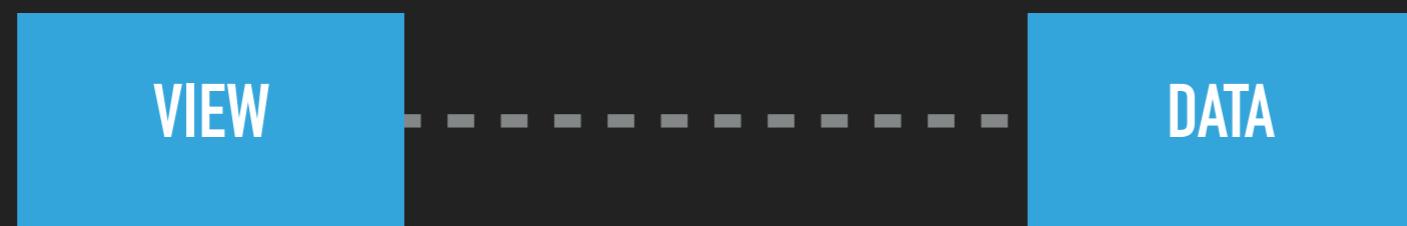
**SWIFTUI**

## DATA

- ▶ Data is the information that drives your UI
  - ▶ Array of model objects -> **List**
  - ▶ Boolean -> **Toggle**

## DATA AS A DEPENDENCY

- ▶ When a view needs to access a piece of data, you create a **dependency** between the view and the data
- ▶ Each time the data changes, you need to update the view



## DATA AS A DEPENDENCY

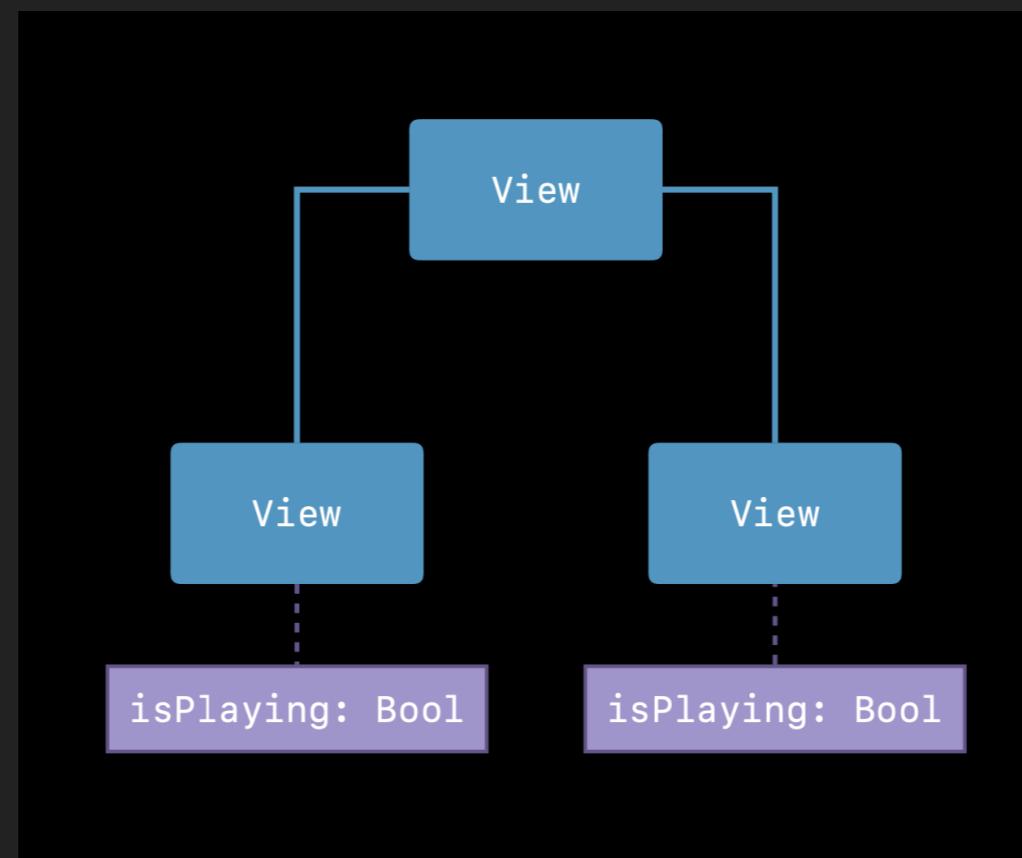
- ▶ With UIKit, keeping view and data in sync requires manual effort
  - ▶ If the data changes, you need to update the view
  - ▶ If the view changes, you need to update the data

## DATA AS A DEPENDENCY

- ▶ With SwiftUI, data dependency is **declarative**
- ▶ You describe the dependency to the framework and the framework handles keeping data and view in sync

# SOURCE OF TRUTH

- ▶ Every piece of data should have a single **source of truth**
- ▶ Keeping duplicate copies of data is a common source of bugs





# SOURCE OF TRUTH

► Example:

- Playback control at the bottom of the screen
- Play button in each row

Castaway



Fresh Air

NPR

+ SUBSCRIBE

My Episodes

All Episodes

## What Happens To The Stuff You Donate?



"Your average thrift store in the United States only sells about one third of the stuff that ends up on its shelves," Adam Minter...

Dec 4, 2019 • 47 min left

## The 'Fundamentally Flawed' War In Afghanistan



After a 3-year legal battle, 'The Washington Post' obtained a trove of government documents, unpublished notes and intervi...

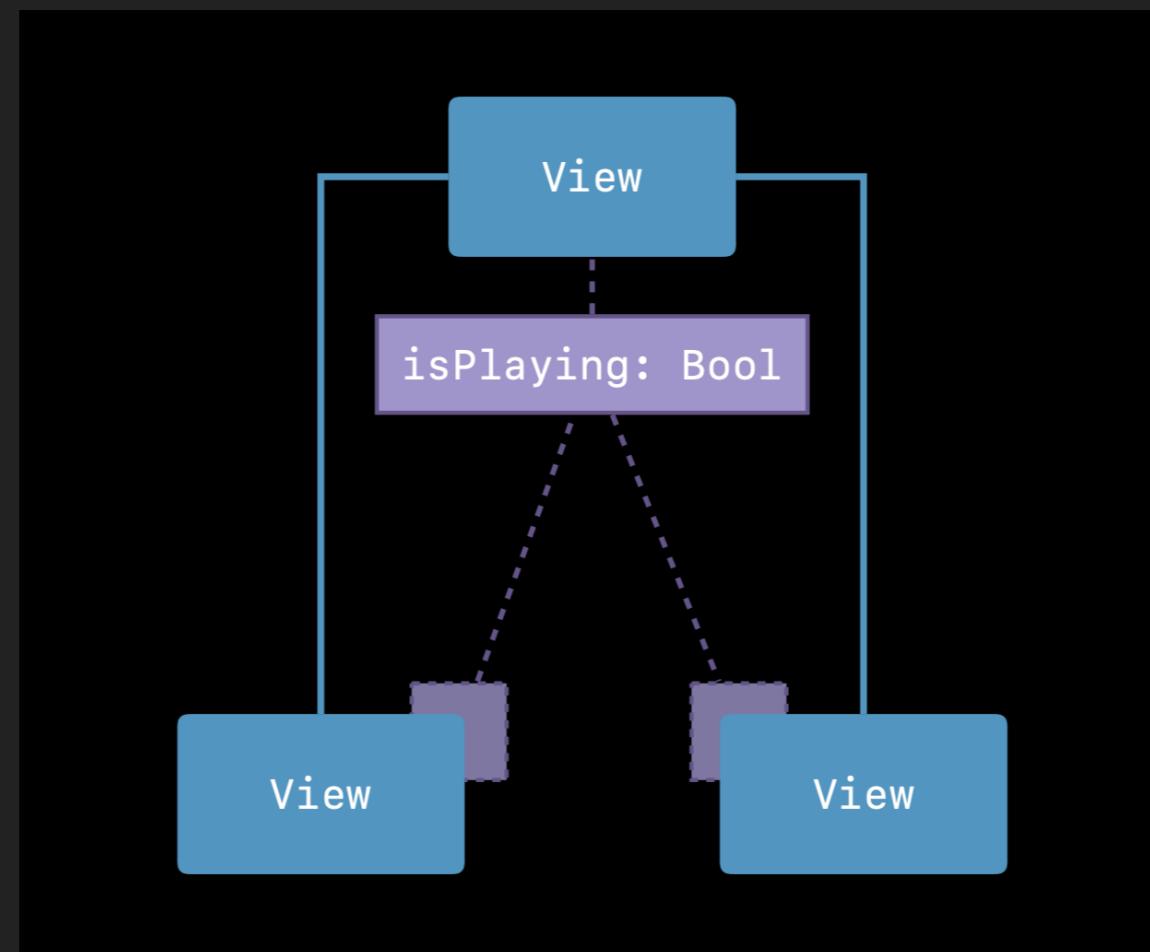
Dec 18, 2019 • 49 min

## 'American Factory' Doc. Filmmakers On Chinese/U.S.



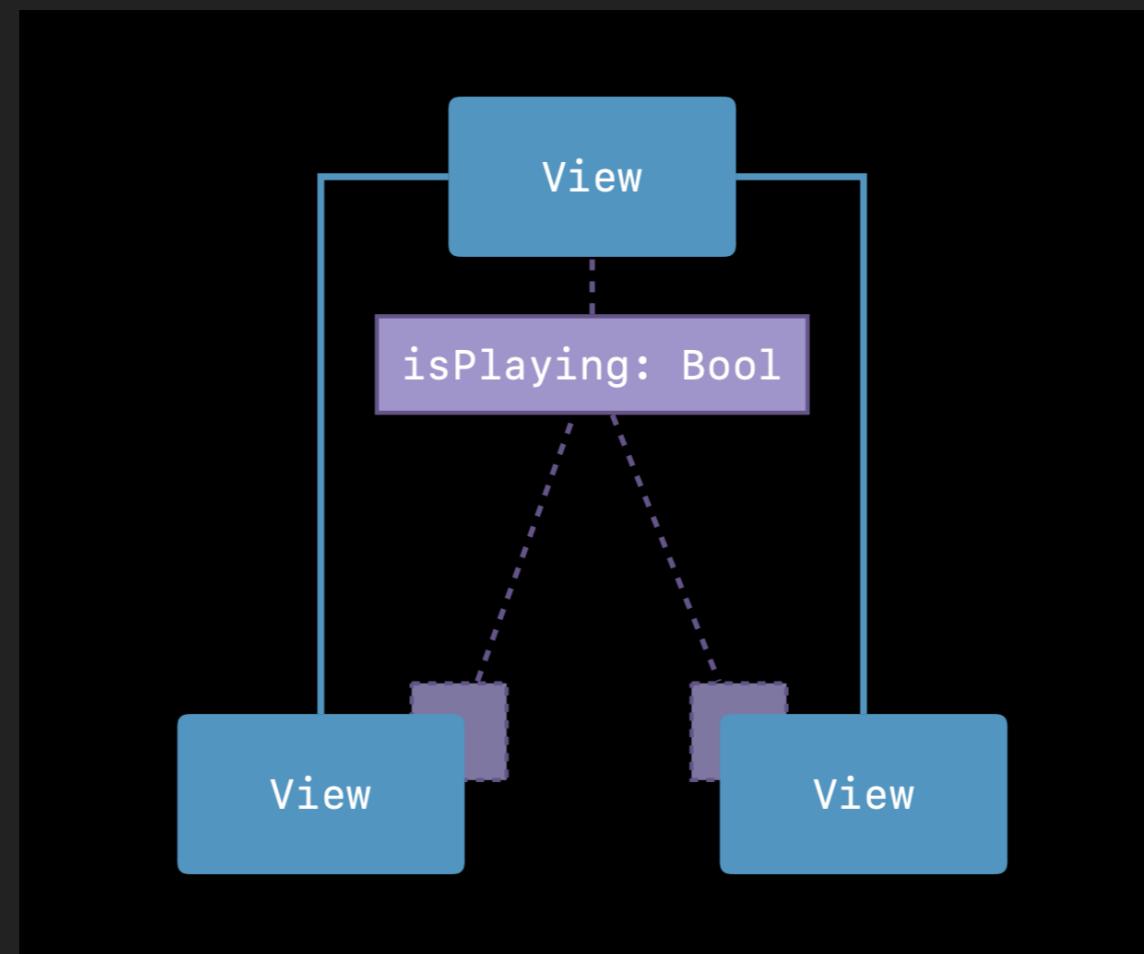
# SOURCE OF TRUTH

- ▶ SwiftUI's solution: maintain a single source of truth in a common ancestor and let two child views have a reference to it



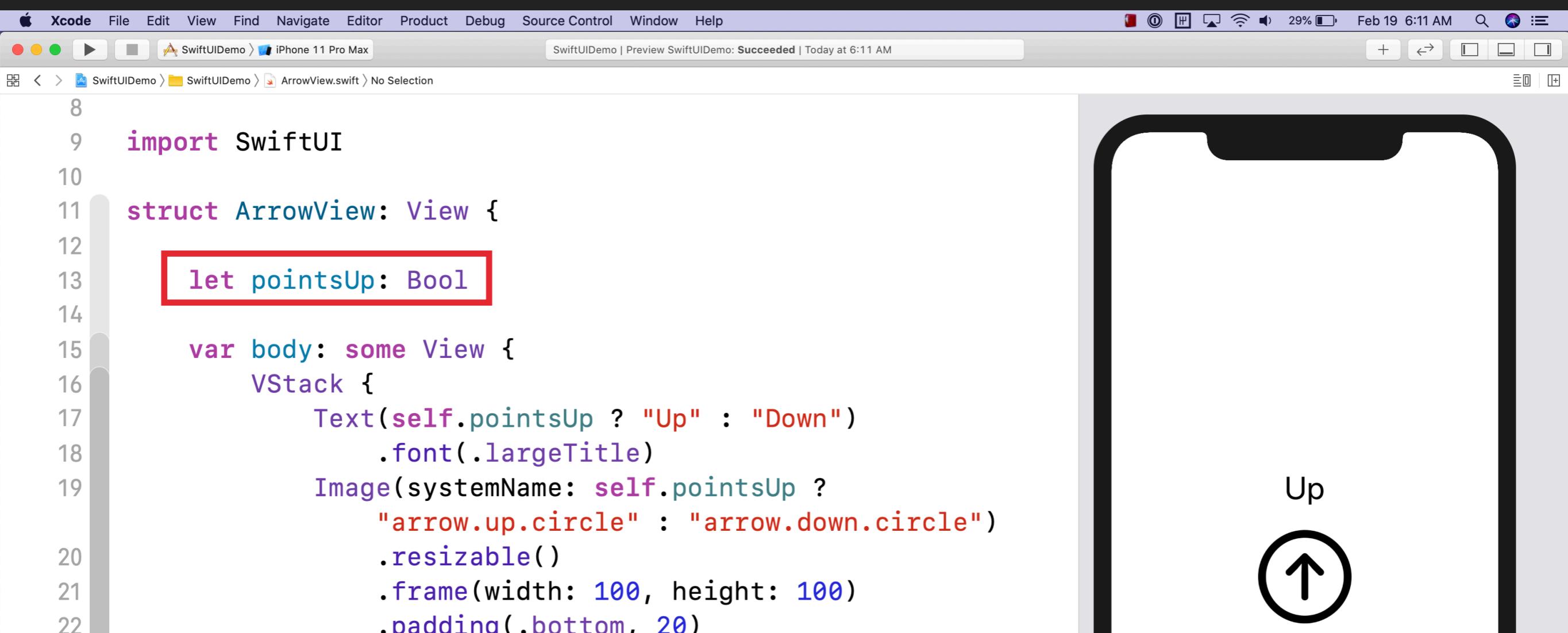
# SOURCE OF TRUTH

- ▶ SwiftUI's solution: maintain a single source of truth in a common ancestor and let two child views have a reference to it



# HANDLING DATA

- ▶ If the view needs read-only access to a piece of data, use a simple Swift property



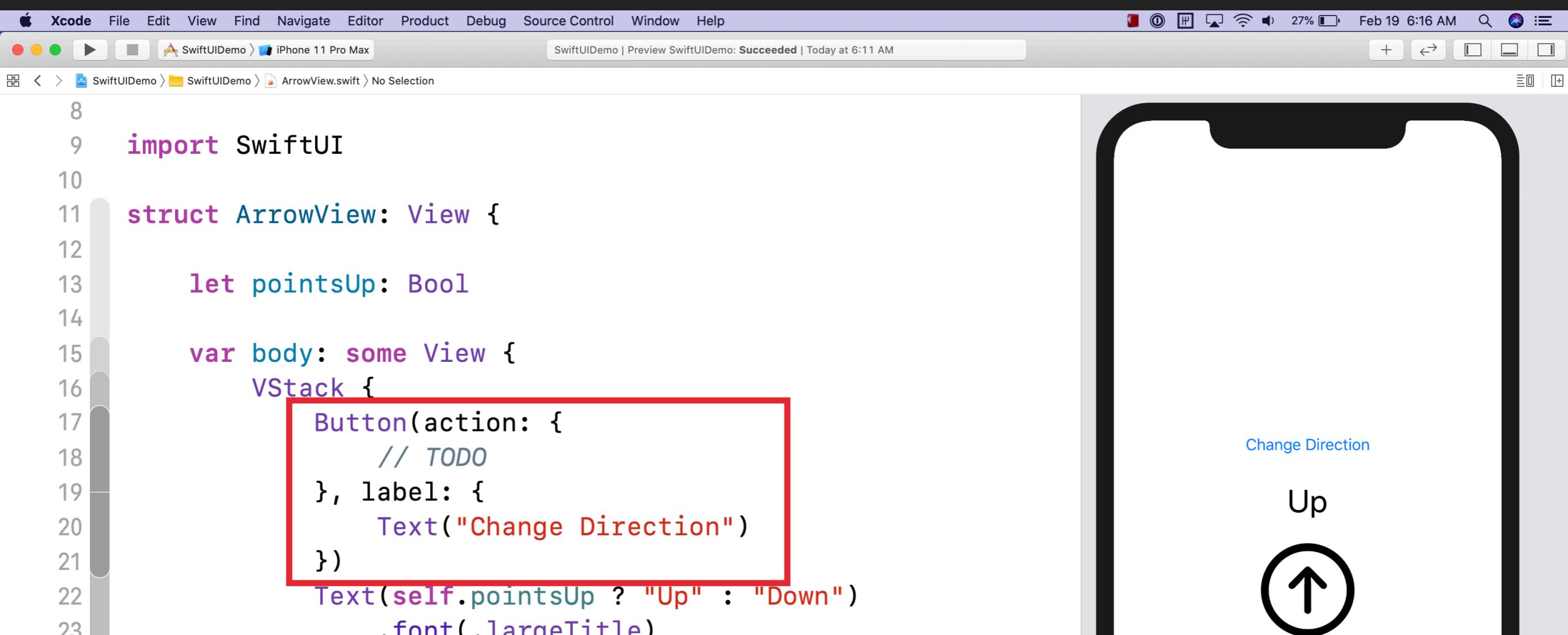
The screenshot shows the Xcode interface with a project named "SwiftUIDemo". The "ArrowView.swift" file is open, displaying the following code:

```
8
9 import SwiftUI
10
11 struct ArrowView: View {
12
13     let pointsUp: Bool
14
15     var body: some View {
16         VStack {
17             Text(self.pointsUp ? "Up" : "Down")
18                 .font(.largeTitle)
19             Image(systemName: self.pointsUp ?
20                   "arrow.up.circle" : "arrow.down.circle")
21                 .resizable()
22                 .frame(width: 100, height: 100)
23                 .padding(.bottom, 20)
```

A red rectangular box highlights the line "let pointsUp: Bool". To the right of the code, a preview of an iPhone 11 Pro Max shows a white screen with a black border. In the center, there is an upward-pointing arrow inside a circle, with the word "Up" written next to it.

# HANDLING DATA

- ▶ Let's add a button to this view and let it determine the direction the arrow is pointing



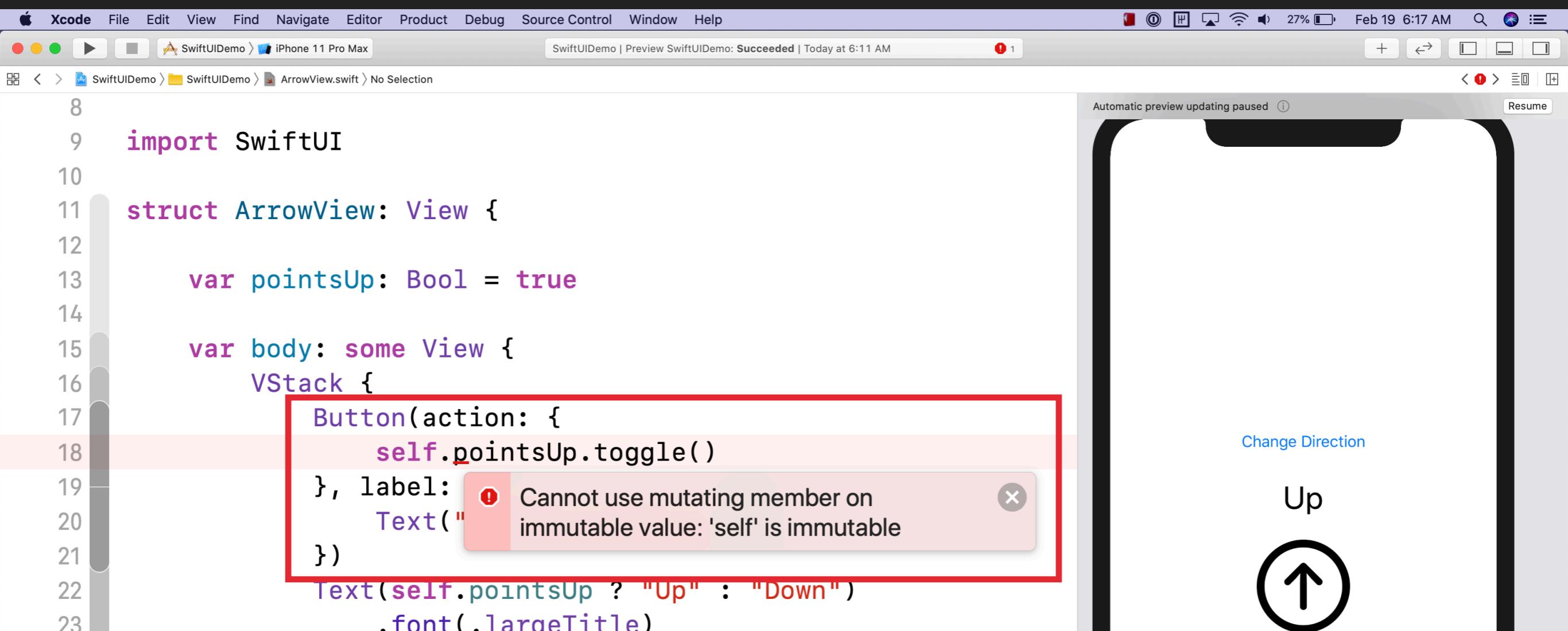
The screenshot shows the Xcode interface with a SwiftUI preview window. The preview displays an iPhone 11 Pro Max screen. On the screen, there is a large black arrow pointing upwards. To the right of the arrow, the text "Up" is displayed above a circular button containing a black upward-pointing arrow icon. Above the preview, the Xcode menu bar shows "File", "Edit", "View", "Find", "Navigate", "Editor", "Product", "Debug", "Source Control", "Window", and "Help". The status bar at the top right indicates the date as "Feb 19" and the time as "6:16 AM". The preview window title is "SwiftUIDemo | Preview SwiftUIDemo: Succeeded | Today at 6:11 AM". The code editor on the left contains the following Swift code:

```
8  
9 import SwiftUI  
10  
11 struct ArrowView: View {  
12  
13     let pointsUp: Bool  
14  
15     var body: some View {  
16         VStack {  
17             Button(action: {  
18                 // TODO  
19             }, label: {  
20                 Text("Change Direction")  
21             })  
22             Text(self.pointsUp ? "Up" : "Down")  
23             .font(.largeTitle)  
24         }  
25     }  
26 }
```

A red rectangular box highlights the `Button` and `Text` blocks from line 17 to 23. The `Text` block contains the string "Change Direction" in red.

# HANDLING DATA

- ▶ Simply changing the boolean in the button's action won't work



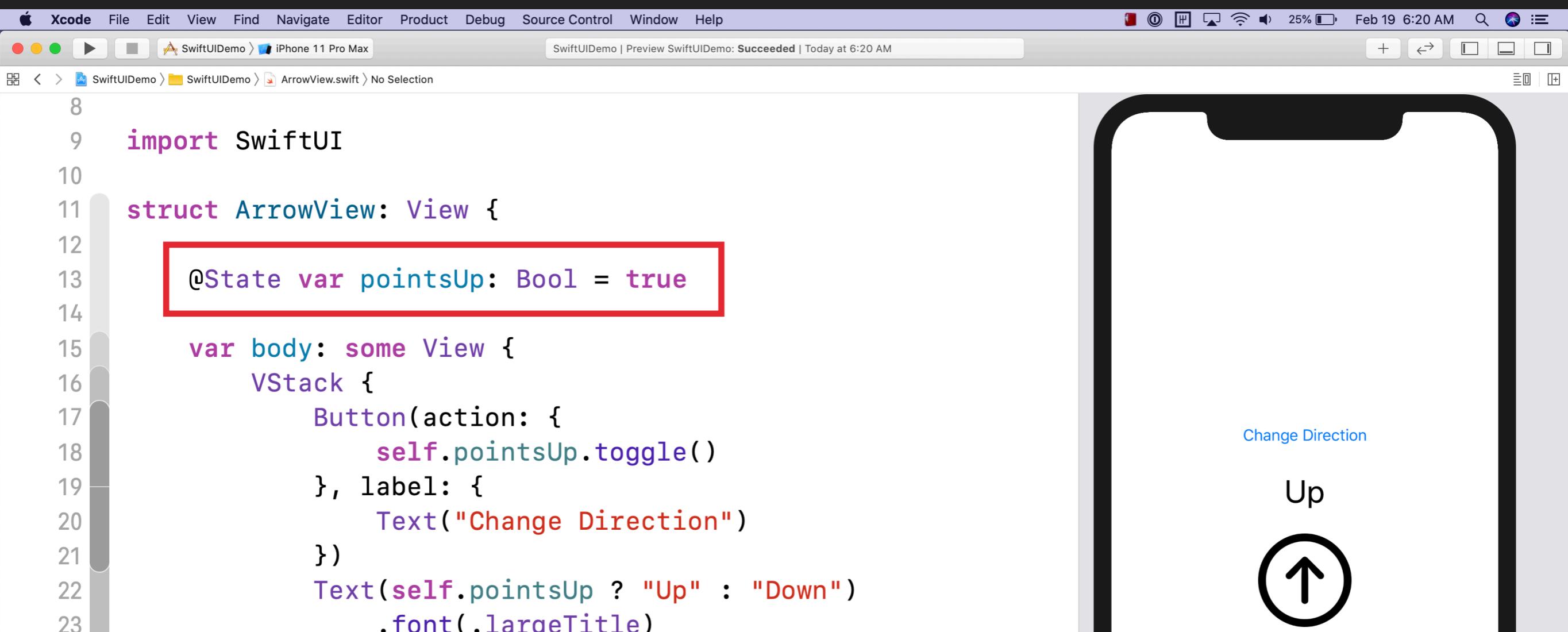
The screenshot shows the Xcode interface with a SwiftUI preview window. The code in the editor is:

```
8
9 import SwiftUI
10
11 struct ArrowView: View {
12
13     var pointsUp: Bool = true
14
15     var body: some View {
16         VStack {
17             Button(action: {
18                 self.pointsUp.toggle()
19             }, label: Text("Up"))
20             Text(self.pointsUp ? "Up" : "Down")
21         }
22     }
23 }
```

A red box highlights the line `self.pointsUp.toggle()`. A tooltip message is displayed over the box: **Cannot use mutating member on immutable value: 'self' is immutable**. The preview window shows a white arrow pointing upwards with the text "Up" next to it.

# HANDLING DATA

- We need to mark it with the `@State` attribute



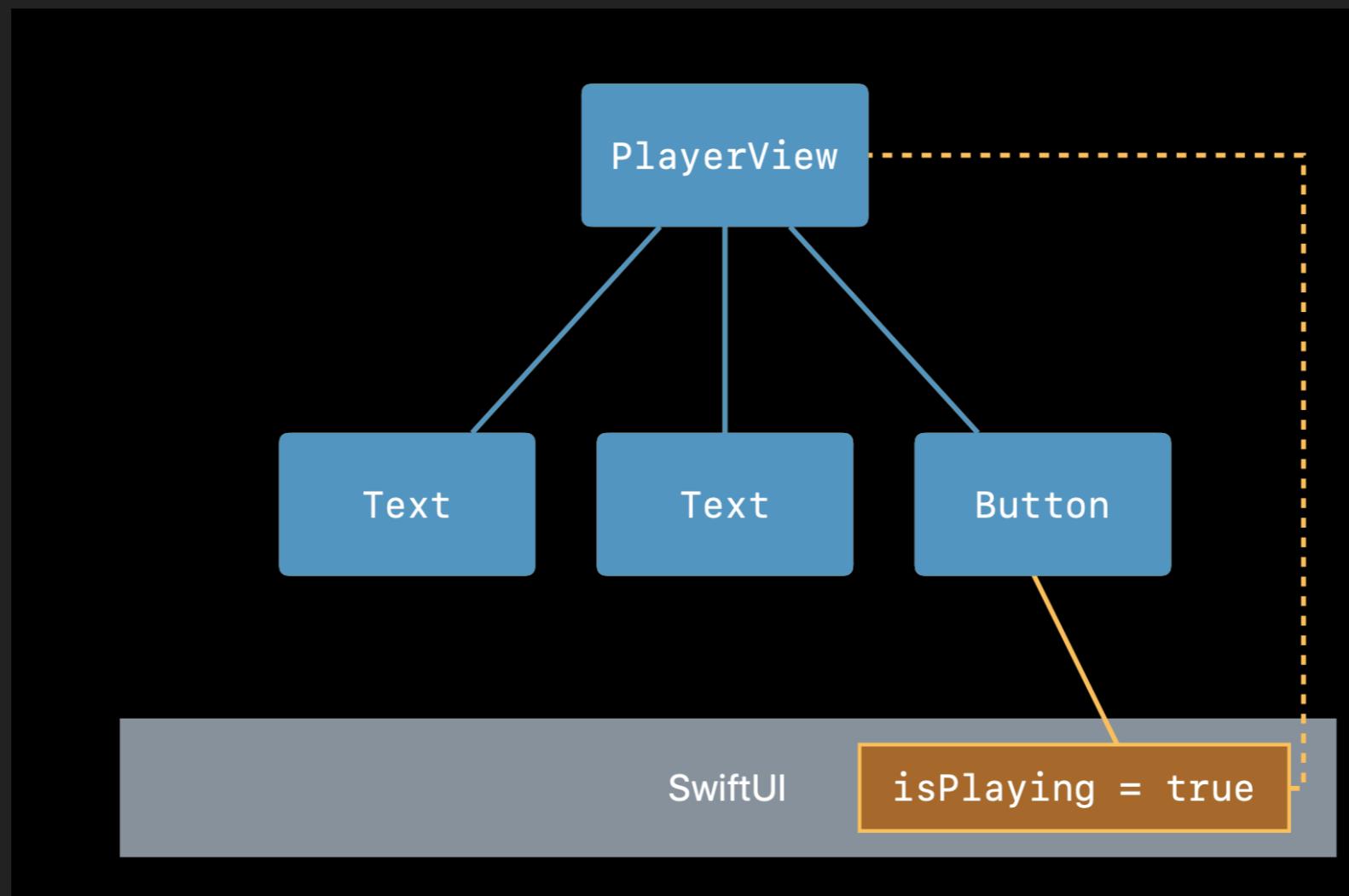
The screenshot shows the Xcode interface with the following details:

- File Navigator:** Shows the project structure: SwiftUIDemo > SwiftUIDemo > ArrowView.swift.
- Editor:** Displays the `ArrowView.swift` file content. A red box highlights the `@State var pointsUp: Bool = true` line.
- Preview:** An iPhone 11 Pro Max preview shows a white screen with a black circular button containing an upward-pointing arrow. The text "Up" is to its right, and the text "Change Direction" is above the button.
- Top Bar:** Shows the Xcode menu bar (File, Edit, View, etc.) and the status bar indicating the date and time.

```
8
9 import SwiftUI
10
11 struct ArrowView: View {
12
13     @State var pointsUp: Bool = true
14
15     var body: some View {
16         VStack {
17             Button(action: {
18                 self.pointsUp.toggle()
19             }, label: {
20                 Text("Change Direction")
21             })
22             Text(self.pointsUp ? "Up" : "Down")
23             .font(.largeTitle)
24         }
25     }
26 }
```

# HANDLING DATA

- When a property marked with `@State` is updated, the change flows down through the view hierarchy



VIEWS ARE A FUNCTION OF  
STATE, NOT OF A SEQUENCE OF  
EVENTS.

# HANDLING DATA

- ▶ Suppose we want to extract button code into its own view

The screenshot shows the Xcode interface with the following details:

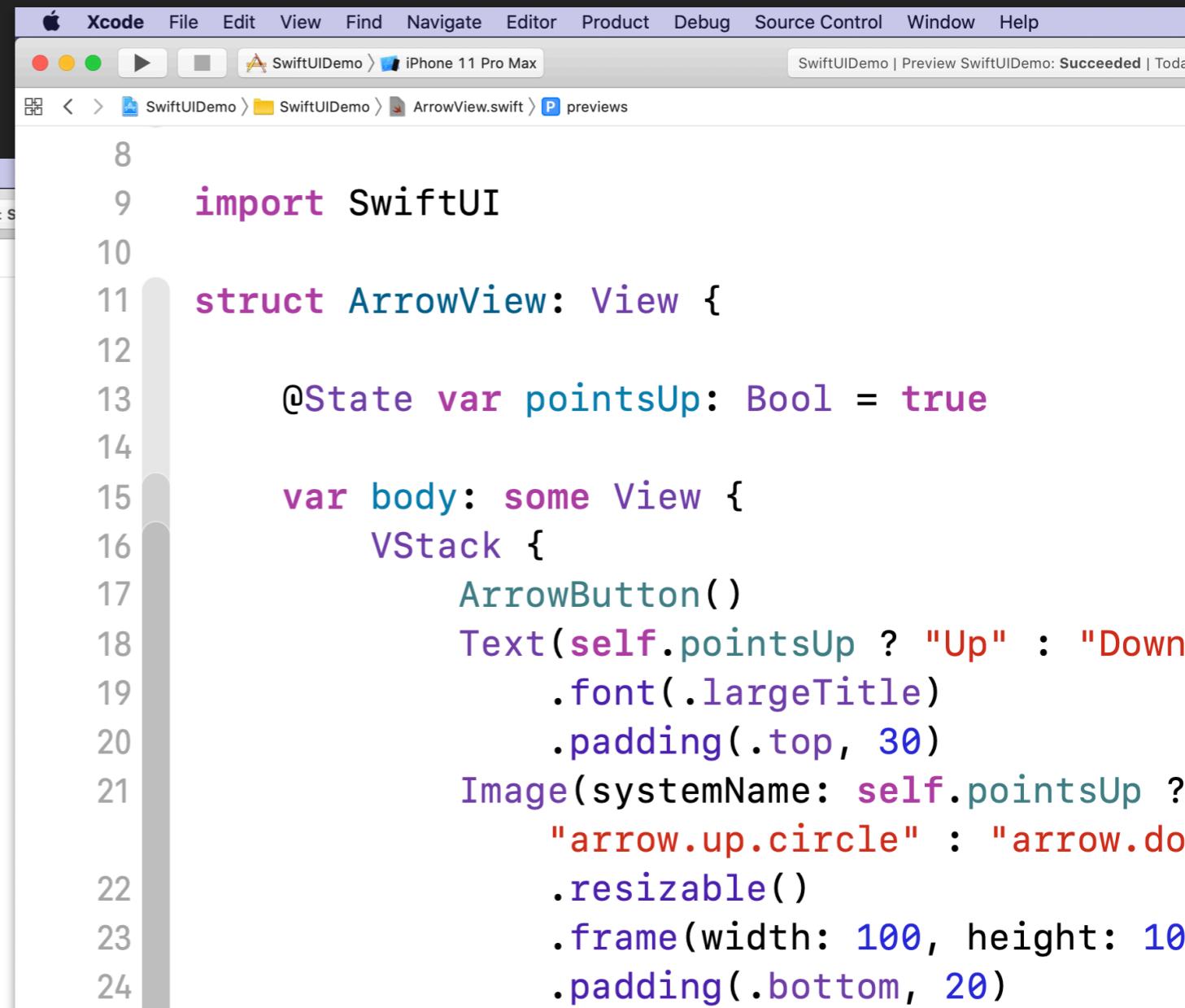
- Top Bar:** Xcode, File, Edit, View, Find, Navigate, Editor, Product, Debug, Source Control, Window, Help.
- Preview Window:** SwiftUIDemo | Preview SwiftUIDemo: Succeeded | Today at 6:32 AM
- Editor:** SwiftUIDemo > iPhone 11 Pro Max
- Code Area:** Shows the `ArrowButton.swift` file content. The code defines a struct `ArrowButton` that contains a state variable `pointsUp` and a body block with a `Button` action that toggles the state and displays a text label indicating the current state ("Point Down" or "Point Up").
- Right Side:** A vertical bar indicates the current state: "Point Down".

```
8
9 import SwiftUI
10
11 struct ArrowButton: View {
12
13     @State var pointsUp: Bool = true
14
15     var body: some View {
16         Button(action: {
17             self.pointsUp.toggle()
18         }, label: {
19             Text(self.pointsUp ? "Point Down" : "Point Up")
20         })
21     }
22 }
```

# HANDLING DATA

- ▶ But wait! Now we have two sources of truth for `pointsUp`

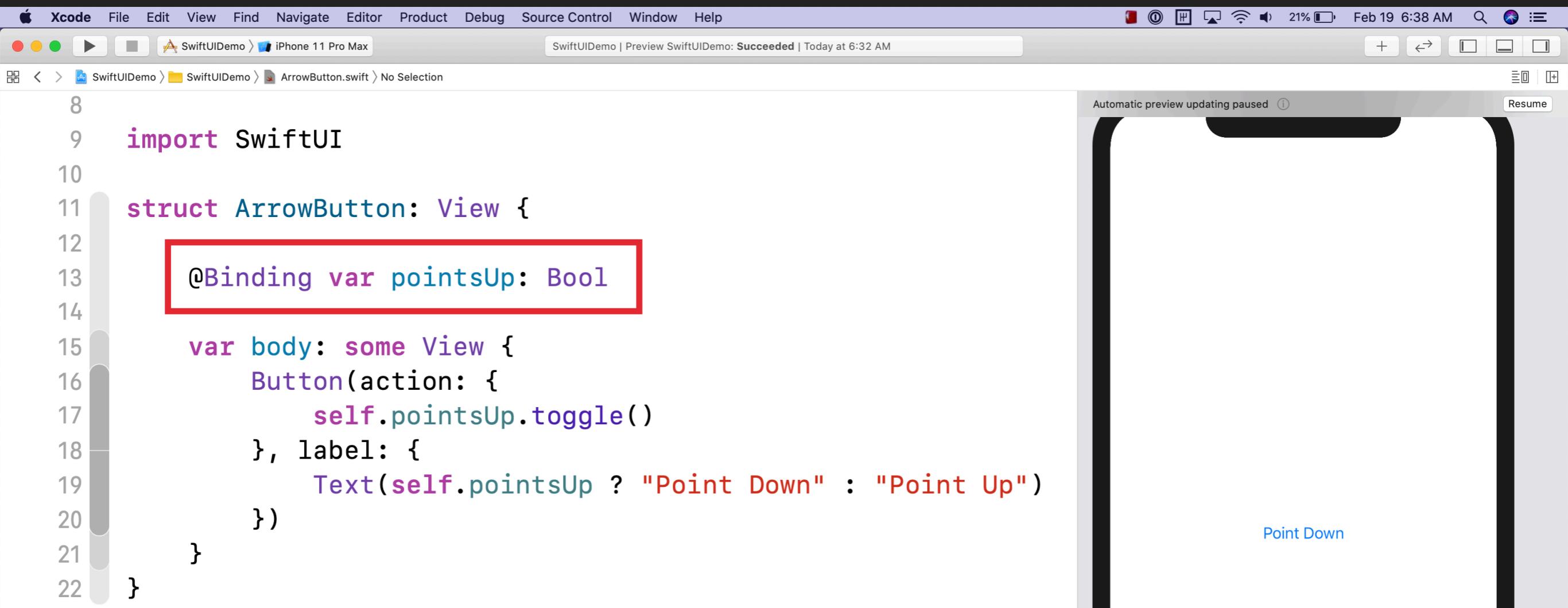
```
8  
9 import SwiftUI  
10  
11 struct ArrowButton: View {  
12  
13     @State var pointsUp: Bool = true  
14  
15     var body: some View {  
16         Button(action: {  
17             self.pointsUp.toggle()  
18         }, label: {  
19             Text(self.pointsUp ? "Point  
20                 ")  
21         })  
22     }  
}
```



```
8  
9 import SwiftUI  
10  
11 struct ArrowView: View {  
12  
13     @State var pointsUp: Bool = true  
14  
15     var body: some View {  
16         VStack {  
17             ArrowButton()  
18             Text(self.pointsUp ? "Up" : "Down")  
19                 .font(.largeTitle)  
20                 .padding(.top, 30)  
21             Image(systemName: self.pointsUp ?  
22                 "arrow.up.circle" : "arrow.down.circle")  
23                 .resizable()  
24                 .frame(width: 100, height: 100)  
25                 .padding(.bottom, 20)
```

# HANDLING DATA

- ▶ Instead we can use `@Binding` to allow a view to read and write a value that it doesn't own



The screenshot shows an Xcode interface with a SwiftUI preview window. The code editor contains the following Swift code:

```
8
9 import SwiftUI
10
11 struct ArrowButton: View {
12
13     @Binding var pointsUp: Bool
14
15     var body: some View {
16         Button(action: {
17             self.pointsUp.toggle()
18         }, label: {
19             Text(self.pointsUp ? "Point Down" : "Point Up")
20         })
21     }
22 }
```

The line `@Binding var pointsUp: Bool` is highlighted with a red rectangle. The preview window shows a black button with the text "Point Down" in blue.

# HANDLING DATA

- ▶ **@State** is useful when:
  - ▶ The view determines the value of the property
  - ▶ The data is a simple type (Boolean, Int, etc.)
- ▶ What if the data is more complex and determined by something external to the view?
  - ▶ Example: Network request

## HANDLING DATA

- ▶ To represent more complex data, create a class that conforms to the `ObservableObject` protocol

```
class GitHubIssues: ObservableObject {  
}
```

# HANDLING DATA

- ▶ Use **@Published** to indicate properties that should cause views to re-render when they change

```
class GitHubIssues: ObservableObject {  
  
    @Published var openIssues: [GitHubIssue] = []  
    @Published var closedIssues: [GitHubIssue] = []  
  
}
```

# HANDLING DATA

- ▶ Whenever `@Published` properties are updated, SwiftUI will automatically re-draw your views with the new data

```
class GitHubIssues: ObservableObject {  
  
    @Published var openIssues: [GitHubIssue] = []  
    @Published var closedIssues: [GitHubIssue] = []  
  
    init() {  
        // Fetch data using URLSession  
    }  
}
```

# HANDLING DATA

- ▶ To reference an `ObservableObject` in a view, use `@ObservedObject`

```
import SwiftUI

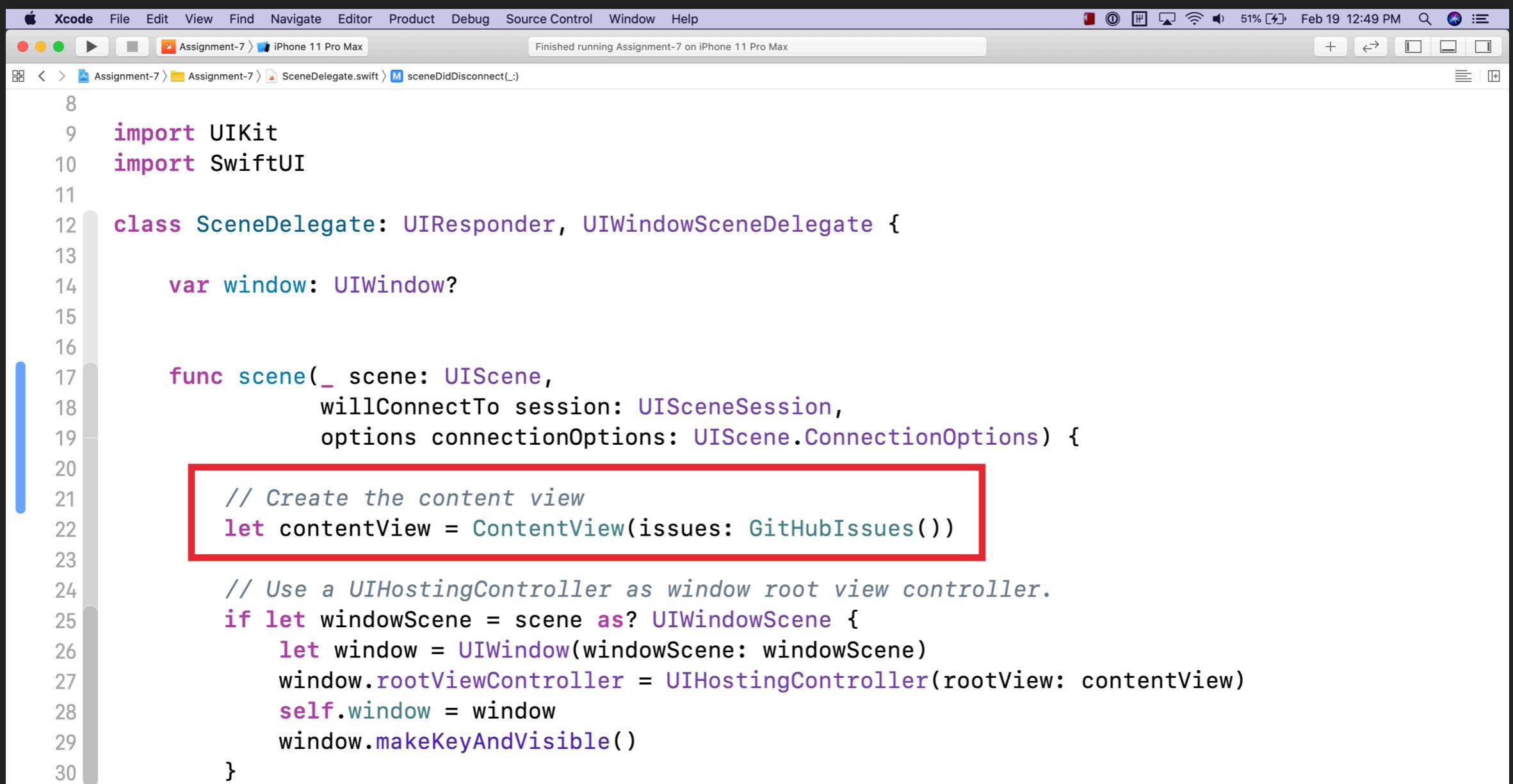
struct ContentView: View {

    @ObservedObject var issues: GitHubIssues

    var body: some View {
        // Define the view's body
    }
}
```

# HANDLING DATA

- ▶ Pass the `ObservableObject` into the view's initializer

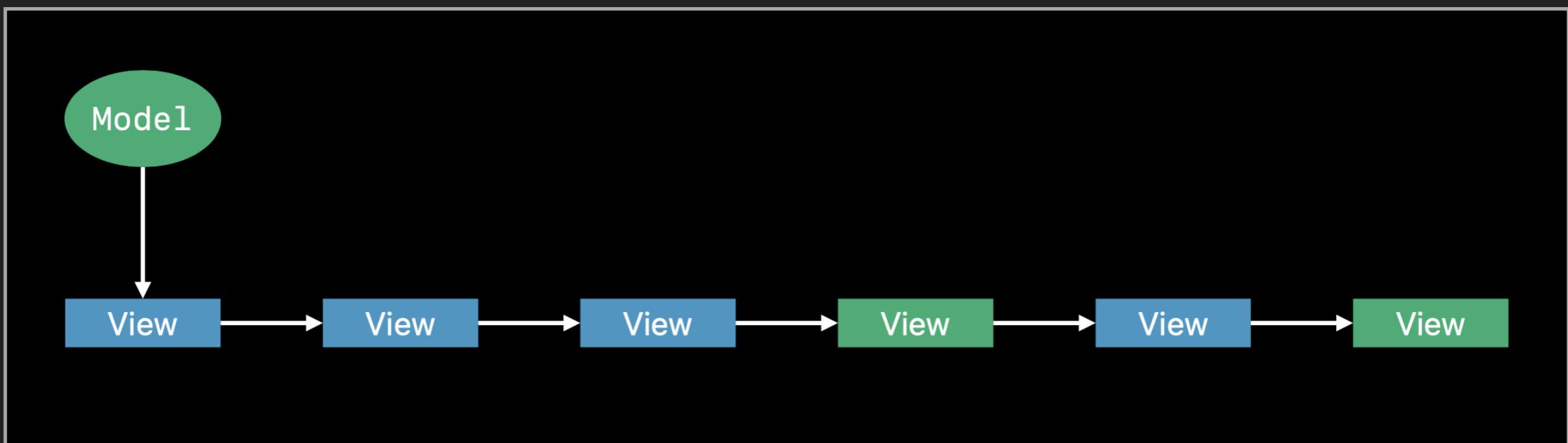


The screenshot shows the Xcode interface with the file `SceneDelegate.swift` open. The code defines a `SceneDelegate` class that conforms to `UIResponder` and `UIWindowSceneDelegate`. It contains a `window` variable and a `func scene` method. The `scene` method creates a `ContentView` instance with `GitHubIssues()` as its parameter, which is highlighted with a red box. The code then uses a `UIHostingController` to set it as the root view controller of the window.

```
8
9 import UIKit
10 import SwiftUI
11
12 class SceneDelegate: UIResponder, UIWindowSceneDelegate {
13
14     var window: UIWindow?
15
16
17     func scene(_ scene: UIScene,
18                willConnectTo session: UISceneSession,
19                options connectionOptions: UIScene.ConnectionOptions) {
20
21         // Create the content view
22         let contentView = ContentView(issues: GitHubIssues())
23
24         // Use a UIHostingController as window root view controller.
25         if let windowScene = scene as? UIWindowScene {
26             let window = UIWindow(windowScene: windowScene)
27             window.rootViewController = UIHostingController(rootView: contentView)
28             self.window = window
29             window.makeKeyAndVisible()
30     }
}
```

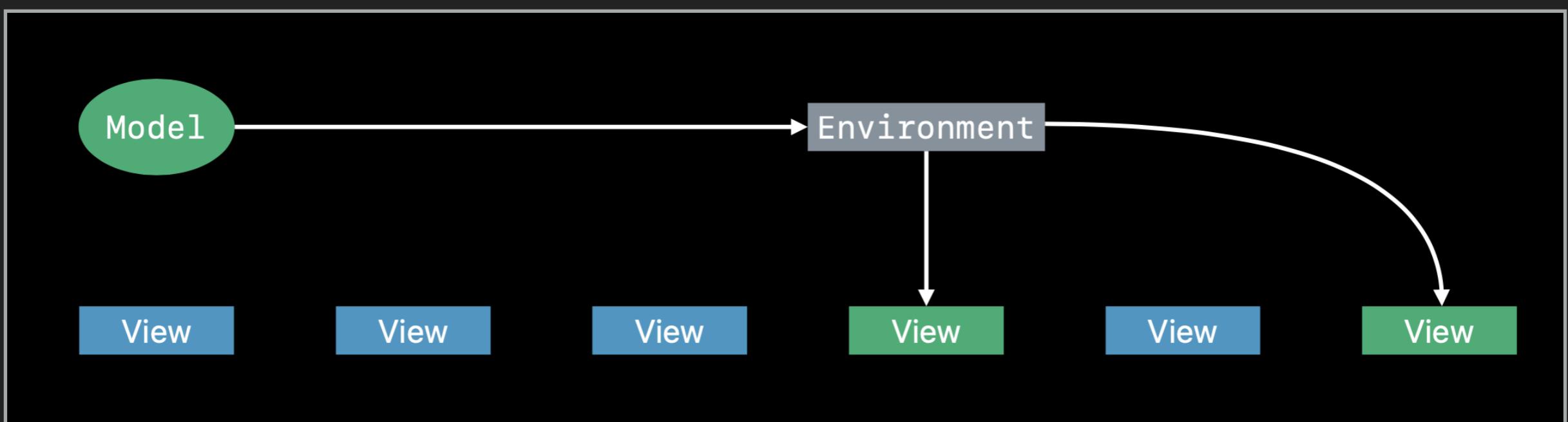
# HANDLING DATA

- ▶ Continue passing the `ObservableObject` into the initializer for any view that requires it



# HANDLING DATA

- ▶ Alternatively, use `@EnvironmentObject` so that any view can access the data indirectly



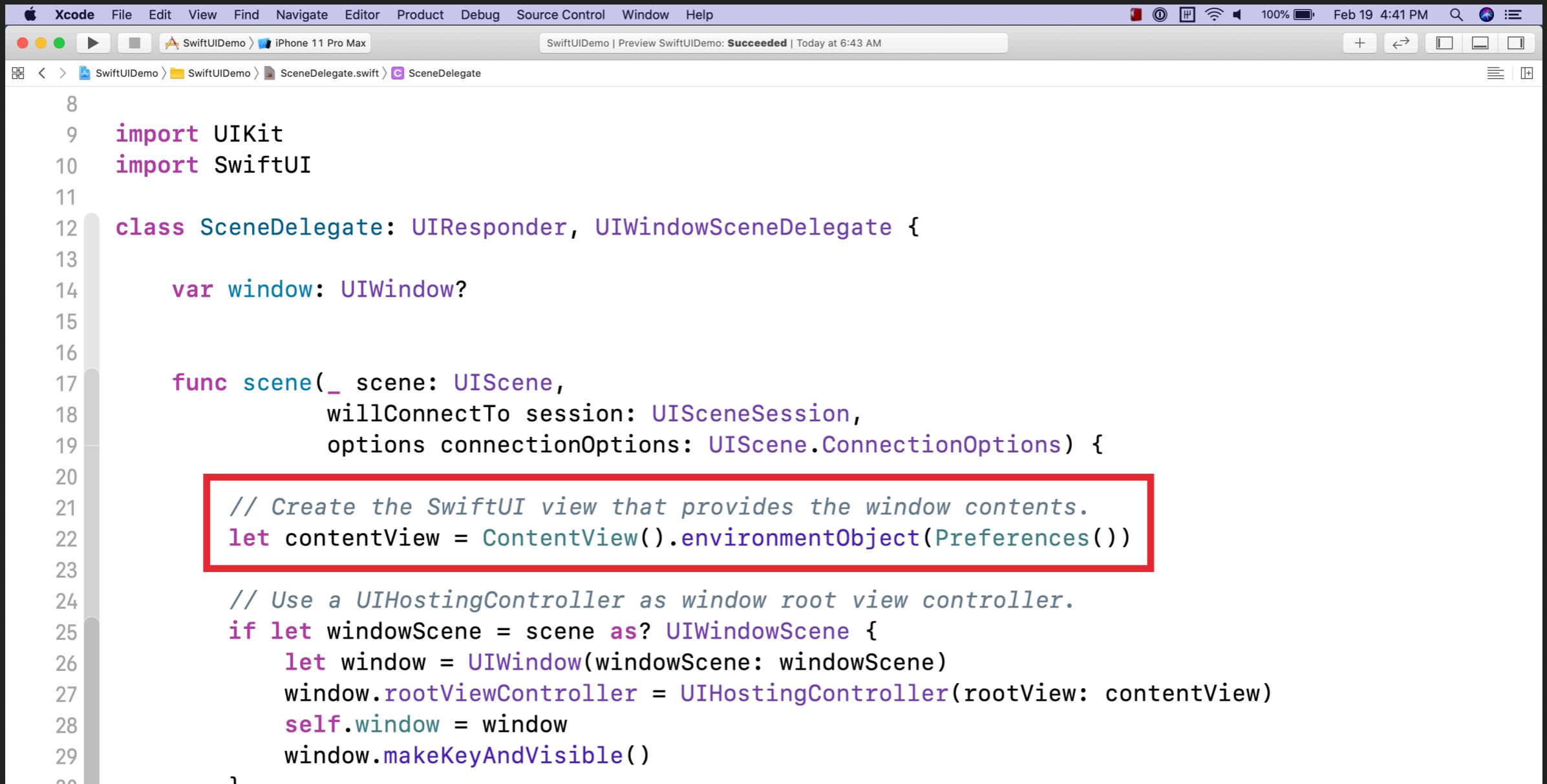
# HANDLING DATA

- ▶ Your `ObservableObject` stays the same

```
class GitHubIssues: ObservableObject {  
  
    @Published var openIssues: [GitHubIssue] = []  
    @Published var closedIssues: [GitHubIssue] = []  
  
    init() {  
        // Fetch data using URLSession  
    }  
}
```

# HANDLING DATA

- ▶ Use the `.environmentObject` method to set the data



```
8
9 import UIKit
10 import SwiftUI
11
12 class SceneDelegate: UIResponder, UIWindowSceneDelegate {
13
14     var window: UIWindow?
15
16
17     func scene(_ scene: UIScene,
18                willConnectTo session: UISceneSession,
19                options connectionOptions: UIScene.ConnectionOptions) {
20
21         // Create the SwiftUI view that provides the window contents.
22         let contentView = ContentView().environmentObject(Preferences())
23
24         // Use a UIHostingController as window root view controller.
25         if let windowScene = scene as? UIWindowScene {
26             let window = UIWindow(windowScene: windowScene)
27             window.rootViewController = UIHostingController(rootView: contentView)
28             self.window = window
29             window.makeKeyAndVisible()
30
31 }
```

## SUMMARY

- ▶ `@State` - simple properties owned by a single view
- ▶ `@ObservedObject` - complex properties shared across several views
- ▶ `@EnvironmentObject` - complex properties accessible to all views

## SUMMARY

- ▶ **@Binding** - gives a view read/write access to a **@State** property that it doesn't own

## SUMMARY

- ▶ **ObservableObject** - protocol that data models must conform to in order to be used with `@ObservedObject` or `@EnvironmentObject`
- ▶ **@Published** - property on an `ObservableObject`
  - ▶ Views using an `@Published` property will re-render each time the property changes

ROUGH EDGES

---

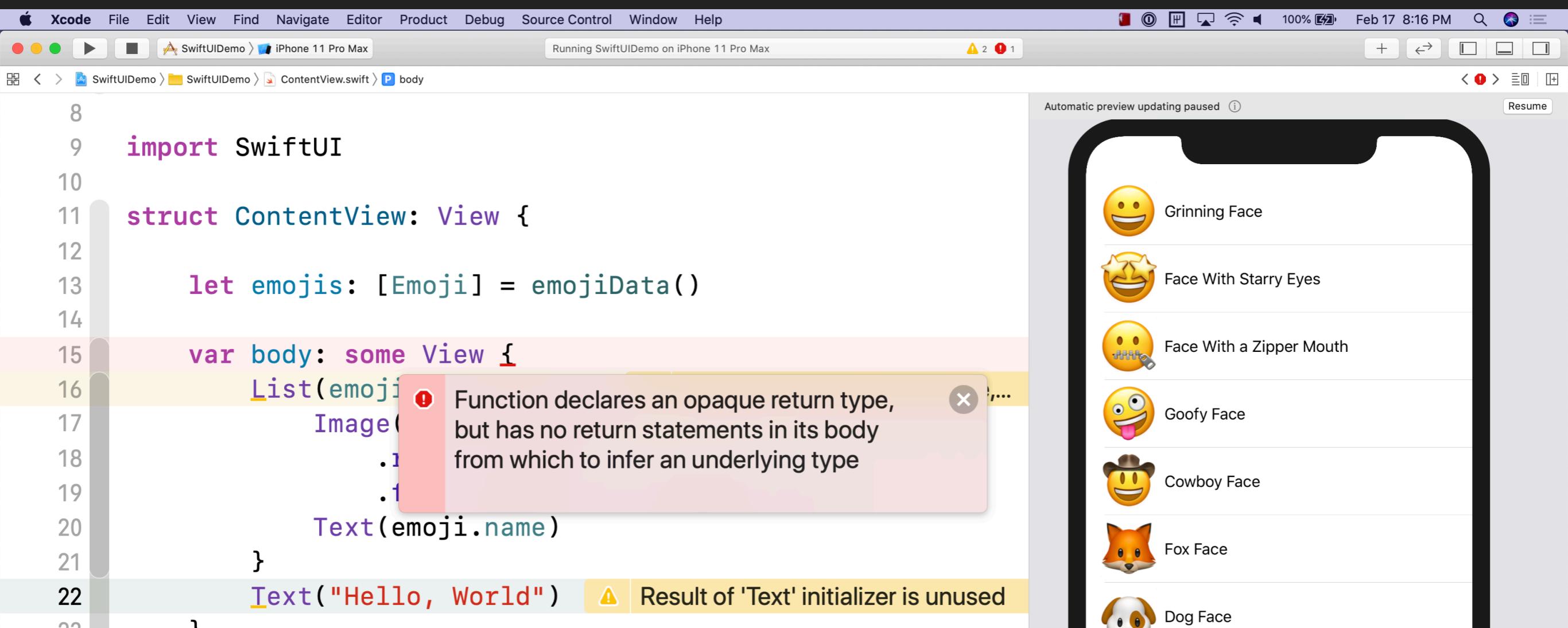
SWIFTUI

## ROUGH EDGES

- ▶ As you're working with SwiftUI, keep in mind that it's a brand new API
- ▶ This means there are a few rough edges

# ROUGH EDGE #1

- ▶ Error messages are sometimes unhelpful
- ▶ They may even appear on the wrong line



The screenshot shows an Xcode workspace with a SwiftUI project named "SwiftUIDemo". The code editor displays the following Swift code:

```
8
9 import SwiftUI
10
11 struct ContentView: View {
12
13     let emojis: [Emoji] = emojiData()
14
15     var body: some View {
16         List(emoji) { emoji in
17             Image(emoji)
18                 .resizable()
19                 .scaledToFit()
20             Text(emoji.name)
21         }
22         Text("Hello, World")
23     }
24 }
```

A tooltip appears over the `List` declaration on line 16, pointing to the first brace of the closure. The tooltip contains the following text:

Function declares an opaque return type,  
but has no return statements in its body  
from which to infer an underlying type

In the bottom right corner of the Xcode interface, there is a preview of the app running on an iPhone 11 Pro Max. The preview shows a list of emoji faces with their names next to them:

Emoji	Name
😊	Grinning Face
😂	Face With Starry Eyes
ZIPPER MOUTH	Face With a Zipper Mouth
😜	Goofy Face
🤠	Cowboy Face
🦊	Fox Face
🐶	Dog Face

## ROUGH EDGE #2

- ▶ The official documentation is spotty at best

The screenshot shows a Mac OS X desktop with a Safari browser window open. The address bar displays 'nooverviewavailable.com'. The main content of the page is a large, bold heading 'No Overview Available.' followed by the subtitle 'A survey of Apple developer documentation.' Below this, there's a section for the 'SwiftUI' framework, which is described as having 4829 documented out of 11901 symbols (40.6% documented). A progress bar indicates the low percentage of documentation. At the bottom, there's a list of several undocumented symbols, each preceded by a '+' sign.

Framework  
**SwiftUI**

4829 / 11901 (40.6%) Documented

7072 Undocumented Symbols

- + !=(\_:\_:) AccessibilityActionKind
- + !=(\_:\_:) AccessibilityChildBehavior
- + !=(\_:\_:) Alignment
- + !=(\_:\_:) AnimatablePair

## ROUGH EDGE #3

- ▶ Live Previews require a lot of CPU power, so you might see your computer start to lag

ASSIGNMENT #7

---

# GITHUB ISSUES

# GITHUB ISSUES

- ▶ Display open and closed GitHub Issues... using SwiftUI!

# GITHUB ISSUES

- ▶ Due Wednesday, February 26 at 5:29pm
- ▶ Post any questions to Slack